University of Pisa and Scuola Superiore Sant'Anna

Master Degree in Computer Science and Networking

Laurea Magistrale in Informatica e Networking

Master Thesis

Cost models for shared memory architectures

Candidate

Fabio Luporini

Supervisor

Prof. Marco Vanneschi

Academic Year 2010/2011

Contents

1 Introduction and thesis goals **1**

2 Background on shared memory architectures **11**
2.1 Processing nodes 13
2.2 Interconnection structures 15
2.2.1 Base latency in networks with wormhole flow control . 17
2.2.2 Base latency in time-slot networks on chip 18
2.2.3 Direct and indirect interconnection structures 18
2.3 Shared memory 21
2.3.1 UMA and NUMA architectures 21
2.3.2 Base and under-load memory access latency 22
2.4 Synchronization mechanisms 23
2.5 Cache coherence 24
2.6 Structured programming on shared memory architectures . . . 26

3 Queueing theory-based cost models **29**
3.1 Elements of Queueing Theory 30
3.1.1 Description and characterization of a queue 30
3.1.2 Notably important queues 33
3.1.3 Networks of queues 35
3.2 Processors-memory system as closed queueing network 37
3.2.1 Formalization of the model 37
3.2.2 Performance analysis of the model 39
3.3 Processors-memory system as client-server model with request-reply behaviour 42
3.3.1 Formalization of the model 42

3.3.2 Assumptions and variants 43
3.3.3 Model resolution 45
3.3.4 A new resolution technique for exponential servers . . . 48
3.3.5 Results 49
3.3.6 On the potential impact of the interconnection network 56
3.3.7 Conclusions 57

4 Stochastic process algebra formalization of client-server models 61
4.1 PEPA: a process algebra for quantitative analysis 63
4.1.1 The PEPA language 64
4.1.2 On the resolution of PEPA models 68
4.2 Fitting general distributions in PEPA terms 68
4.3 A PEPA model of client-server systems with request-reply behaviour 69
4.3.1 The general model 69
4.3.2 Applying the model to the processors-memory system . 70
4.3.3 Model resolution 71
4.4 Quantitative comparison with respect to other resolution techniques 72
4.5 An example: load-dependent service times in a client-server model 77

5 Applying the cost model to a concrete architecture 79
5.1 Tilera TILEPro64 architecture overview 79
5.2 Memory access latency in Tilera architectures 82
5.2.1 Methodology 82
5.2.2 Base latency 83
5.2.3 Under-load latency 85
5.3 Performance evaluation of a parallel application 86

6 Conclusions 93
6.1 Summary 93
6.2 Further work 95

References 97

Chapter 1

Introduction and thesis goals

Structured parallel programming on shared memory architectures

High Performance Computing (HPC) deals with hardware-software architectures and applications that demand high processing bandwidth and low response times. Shared memory multiprocessors (SMP, NUMA) are a notable example of HPC systems. To be exploited efficiently, these systems require the development of parallel applications characterized by high levels of performance and scalability.

Currently, an important technological revolution is taking place: Chip MultiProcessors (CMP), simply known as multi-cores, are replacing uniprocessor-based CPUs for both the scientific and the commercial market. They can be considered multiprocessors integrated on a single chip, thus many theoretical results found for classical shared memory architectures are also valid for multi-cores. According to new interpretations of the "Moore's law", the number of cores on a single chip is expected to double every 1.5 years. It is clear that so much computational power can be used at best only resorting on parallel processing.

The spread of a parallel programming methodology is acquiring a deep importance in the scientific community. We advocate that parallel applications should be written according to a structured approach, without being influenced by the characteristics of the specific architecture. In *structured parallel programming* (or *skeleton* based parallel programming) a limited set of *parallel paradigms* (skeletons) is used to express the structure the parallel

application. Properties like *simplicity* and *composability*, as well as *parametric cost models*, make structured parallel programming a very attractive approach to dominate the complexity inherent the design and the evaluation of parallel and distributed application.

Despite the theory behind structured parallel programming is quite solid, the gap with respect to shared memory architectures is still wide. Currently there is no way to solve the *performance predictability* problem. The sharing of firmware resources, especially the memory hierarchy, surely optimize the global architecture performance, but at the same time makes single flow performance less predictable, due to the contention for these resources (the so called "conflicts" among processing nodes for a shared unit). This has deep implications in the development of a parallel program, because the only way to have at least a rough idea of its performance, is to look at its execution time. On top of that, performance may change even dramatically moving the application from a target architecture to another one, leading to the *performance portability* problem. Actually, some theoretical work that address these problems exists, but it is still far incomplete for being applied in practical scenarios.

The goal of the thesis falls in this area. We address the *performance predictability* problem for shared memory architectures, with particular emphasis on state-of-the-art multi-cores.

Client-server-based cost models for shared memory architectures

A critical problem in shared memory architectures is to predict in which measure the limited memory bandwidth will influence the application performance. To answer this question a possible approach is to estimate the so called *under-load memory access latency* R_Q, that is the average time to access the main memory subjected to the workload of a parallel application. R_Q measures the ability of the system to execute a certain amount of instructions in presence of *memory conflicts*. The importance of R_Q is crucial and will be explained in details throughout the thesis. However, for now it is sufficient to know that *application cost models*, i.e. cost models that measure the real application bandwidth and/or completion time, will be defined on

top of R_Q.

To predict R_Q, a solution based on Queueing Theory concepts is proposed in [20]. Consider a system in which a set of N client modules $C_1, C_2, ..., C_N$ send requests to a server module S and need to wait for an explicit reply in order to continue their elaboration. A queue Q models the fact that requests could experience a certain waiting time before being handled. An example of this scheme is shown in Figure 1.1.

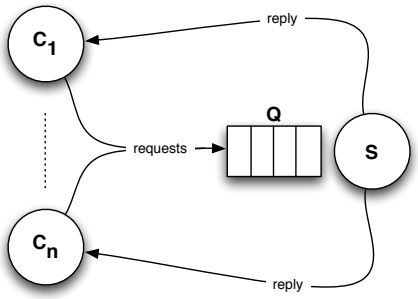


Figure 1.1: Client-server system with request-reply behaviour.

This model fits very well in shared memory architectures: $C_1, C_2, ..., C_N$ are processing nodes that issue memory requests, while S is a memory module. The average response time of S is just R_Q. To derive R_Q, the following values must be known in advance ("constants of the system").

- Let T_P be the mean time between two consecutive accesses of the same processing node to the same memory module (i.e. average time between two consecutive cache faults). We assume that T_P be the mean value of an exponential distributed random variable. We also assume *homogeneous* clients, that is all processing nodes have the same T_P.

- Let p the number of processing nodes that share a specific memory module.

- Let T_S be the mean service time of S, that is the average time between the executions of two consecutive memory requests. For now, we assume T_S be the mean value of an exponential distributed random variable.

Notice that T_P is an *algorithm-dependent* parameter, while T_S is an *architecture-dependent* one. On the other hand, we will see that p depends on both the application and the architecture.

We also introduce the variables of the systems, that is those values that we are interested to determine

- Let T_{cl} be the average time required to issue a request to the memory system and obtaining the reply. This is also called the *client service time*.
- We also know about R_Q, that is the under-load memory access latency.
- Let ρ be the *utilization factor* of the system, such that $\rho \in [0, 1)$. It expresses the congestion degree of the server; for instance, low values of ρ means that the server is capable of handling requests without introducing meaningful overhead. The fact that ρ is always less than 1 implies that the system has a *self-stabilizing* behaviour, that is a *steady-state solution can always be derived*. We will formally justify this aspect in chapters to come, by exploiting concepts of Markovian Theory.
- Let T_A be the average interarrival time at S.

These parameters represent the *cost model of a shared memory architecture* and to determine their value we will need appropriate *resolution techniques*.

The parameters and the semantics of this basic client-server system can be refined according to the specific scenario that we have to model. Notable examples are the following

- The random variable distributions can be different from the exponential one. For instance, the server service time is *deterministic* in case the memory system takes a constant amount of time to retrieve the requested data.
- A hierarchical structure of servers can be used in place of the single server. For instance, if a request cannot be handled by the server i, then it is forwarded to the server $i+1$ of the hierarchy. For instance, we will see that this model fits very well for some state-of-the-art multicores, where one or more levels of caches are shared.

- Each node of the system could be both a client and a server at the same time, even if the interaction remains of type request-reply.
- The server service time is load-dependent, that is it depends on the number of requests currently in the queue.

Throughout the thesis we will apply these models in disparate contexts. Clearly, with so many degrees of freedom, determining appropriate resolution techniques can be a quite complex task.

We will focus on two types of resolution techniques: *analytical* and *numerical* ones. The accuracy of the different techniques will be compared against *direct experimentation* (e.g. real architecture's simulations, queueing networks simulations, etc). On the other hand, we will not consider simulations as a valid alternative for evaluating the architectural cost model, since it may be time-consuming, difficult (in terms of the flexibility of the available tools) and even lack generality.

Analytical resolution techniques for simple client-server models

In [20] the following system of equations is proposed as solution of the client-server system.

$$\begin{cases} T_{cl} = T_P + R_Q \\ R_Q = W_Q(T_s, \rho) + t_{a0} \\ \rho = \frac{T_S}{T_A} \\ T_A = \frac{T_{cl}}{p} \end{cases} \tag{1.1}$$

The behaviour of a client is cyclic: a request is generated every $T_{cl} = T_P + R_Q$ clock cycles. Once known T_{cl}, T_A is determined by resorting on the *aggregate interarrival theorem*. For now, it is sufficient to know that thanks to this theorem we can state $T_A = \frac{T_{cl}}{p}$. Assuming that T_A is the mean value of an exponential random variable, the steady state condition of the system is characterized by $\rho = \frac{T_S}{T_A}$. Finally, R_Q is simply given by the average waiting time W_Q at the queue plus a constant known in advance, which

is the base latency t_{a0} (service latency in absence of conflicts). Different expressions of W_Q can be used according to the Queueing Theory. This resolution technique enables *qualitative* reasoning. For example, assume that T_P increases: consequently, also T_A increases. Thus ρ decreases and finally R_Q decreases. Notice that T_{cl} is subjected to a *feedback effect*: from one hand it tends to increase (T_P was increasing), but at the same time the decrement of R_Q inhibits this progress (feedback effect).

This is an *approximate resolution technique* because it relies on specific assumptions and simplifications. In this perspective, throughout the thesis we will address two aspects. Firstly, we will validate the goodness of the approximation against direct experimentation (simulations of client-server networks). We will see that the results are in general fairly acceptable, except for some critical workloads for which R_Q will be overestimated. To solve this problem, we will propose a new resolution technique, still based upon simple Markovian and Queueing Theory concepts.

The new resolution technique improves the evaluation of the interarrival time at the server. The intuition is that the population of the system is *constant*, thus the interarrival frequency cannot be constant as we had implicitly assumed. Rather, the frequency of arrivals λ_i will be *proportional* to the number of clients i ($0 \leq i \leq p$) that are neither queued nor in service, that is $\lambda_i = i\lambda$. Even this technique will be characterized by a simple closed form solution.

Finally, notice two peculiarities of the cost model we are formalizing: it is *simple*, from both the conceptual and the mathematical point of view, and provide *approximate, yet fairly acceptable, results*. And it is well-known that a cost model should be always characterized by these two properties.

Numerical resolution techniques for extended client-server models

Previously, we mentioned the need of extending the semantics of the basic client-server model because of the complex nature of shared memory architectures. The modelling of hierarchical systems, as well as different service disciplines, are two notable examples. Clearly, to solve the new models we will need appropriate resolution techniques, or at least to modify the ones

we already introduced. The problem is to determine a trade-off between the complexity of the resolution technique and the quality of the approximation. In light of this, to evaluate R_Q we propose to use *numerical* resolution techniques, in place of the analytical ones.

The idea is to describe the client-server model at the level of Markov Chains. There are a lot of resolution methods for moderately sized Continuous Time Markov Chain (CTMC) models, while iterative techniques exist for huge sized models. Since Markov processes can be difficult to construct, we will exploit, as intermediate description language, the *stochastic process algebra* $PEPA$ (Performance Evaluation Process Algebra).

We advocate that the flexibility and the expressiveness of PEPA, as well as its formal background theory, fits very well in the specific context of shared memory architectures. More in general, using PEPA implies the convergence of two different research fields, namely *parallel computing* and *semantics*. Despite PEPA has been used with success in many different disciplines (physics, chemistry, biology and clearly computer science as well), as far as we know this is the first attempt to exploit it for performance evaluation in parallel computing.

In this context, the objectives of the thesis are the following:

- study the effectiveness of the PEPA approach. PEPA is in general a powerful tool, but it really fits in our area of research?

- a quantitative analysis of a PEPA client-server system. A lot of tools are available to automatically solve and evaluate PEPA models. We want to stress the fact that since PEPA is built on top of Markovian Theory, the resolution techniques will be the very traditional ones adopted for computing the steady-state regime of Markov Chains.

Direct experimentation

In the third part of the thesis we in-depth study a state-of-the-art multi-core architecture, the Tilera TILE64. At the end of this study, we will provide and validate a cost model for this architecture.

When studying a concrete architecture, a typical problem is that technical information are not sufficient to instantiate some kind of performance anal-

ysis (in particular a detailed architectural cost model). In the context of the Tile64, we will face the problem of analysing the memory system. Differently from what is typically assumed in literature, the time that a memory module takes to retrieve a certain data block is *not* constant. The memory system takes fully advantage of some properties (e.g. spacial- and time-locality in accessing pages) to improve its global performance, from both the bandwidth and the latency point of view. In order to exploit at best these properties, the Tile64's memory interface implements a smart scheduling algorithm to re-order outstanding memory requests, instead of forwarding them to the memory module in a classical FIFO manner. Intuitively, the larger the number of outstanding memory requests, the better will be the result of the scheduling algorithm and consequently lower the exhibited service time of the memory. This behaviour has deep implications in the definition of the cost model, because the client-server system cannot be applied as it stands (there is no immediate ways to model the service time distribution of the memory). We will see that a load-dependent semantics must be introduced to match the experimental results. In this context, the flexibility of PEPA will play a key role.

Once defined, the Tile64 architectural cost model will be validated against direct experimentation, that is by executing a simple parallel application on the Tile64 simulator.

Structure of the document

In this document we deal with the aforementioned topics.

1. In Chapter 2 a summary of the main concepts about shared memory architectures is presented. Particular emphasis is reserved to multi-cores systems. All aspects like processing node, memory organization, interconnection networks, cache coherence as well as a brief on the parallel programming methodology are treated.

2. Chapter 3 deals with analytical resolution techniques for client-server models. A first part reviews some basic Queueing Theory concepts. Then, two resolution techniques are formalized, compared and validated against experimental results.

3. The stochastic process algebra PEPA is introduced in Chapter 4. A methodology for estimating the under-load memory access latency by resorting on PEPA client-server models is formalized. The quantitative analysis of the model is validated against experimental results.

4. In Chapter 5 the Tile64 multi-core architecture is in-depth studied. An architectural cost model is proposed and validated. PEPA will be exploited to evaluate the cost model.

Chapter 2

Background on shared memory architectures

In this chapter we describe the main concepts about a class of parallel *Multiple Instruction Stream Multiple Data Stream* (MIMD) architectures: *multiprocessors* and *multi-cores* exploiting parallelism at processes level [20, 8, 17]. These architectures will be object of analysis throughout the thesis.

Multiprocessors in a nutshell At first sight, a multiprocessor can be seen as a set of *processing nodes* that share one or more levels of memory hierarchy and are able to exchange firmware messages along an *interconnection structure*. A logical schema of a multiprocessor architecture is shown in Figure 2.1. Processing nodes are *general purpose* CPUs, possibly with a local memory and some I/O units, while the interconnection structure is usually a trade off between performance and cost of the interconnection. The *shared memory* peculiarity means that any CPU is able to address any location of the physical memory. In other words, the result of the translation from logical addresses to physical ones can be any location of main memory. Moreover, lower levels of the memory hierarchy can be shared. The *firmware messages* that flow in the interconnection network can be either shared memory access requests/replies or explicit interprocessor communication. It is worthwhile to stress the fact that these are low level messages, thus they must not be confused with the ones at process level.

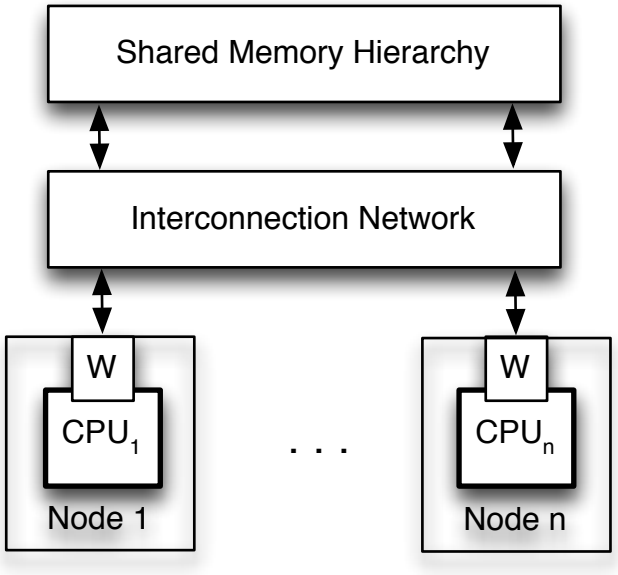


Figure 2.1: Simplified view of a shared memory architecture

State of the art: multi-cores and parallel programming Multi-cores, or *Chip MultiProcessor* (CMP), can be considered shared memory multiprocessor integrated on a single chip. Therefore many results found for multiprocessor architectures are also valid for multi-cores, especially when the number of cores on the same chip is relatively low. The number of cores is expected to double every two years, according to the Moore's law applied to the number of cores on chip. The trend is to substitute few complex and power-consuming CPUs, with many smaller and simpler CPUs that can deliver better performance per watt. Unfortunately, the gap between these architecture and the software is still wide. In spite of the relevant architectural changes, the actual programming tools are still low-level, and this makes parallel programming a rather complex task. Further, performance prediction and performance portability are missing or are still in the infants. The challenge of parallel programming on multi-core architectures could be accomplished resorting on a *structured methodology* and on *performance cost models* [20].

Performance degradation The main problem in shared memory architectures is the *performance degradation* due to the contention for shared resources, especially the memory hierarchy. In the same time interval, a group of processes may access the shared memory, either for reading/writing program's data or for executing concurrency mechanisms, causing *memory conflicts*. Therefore, the effectiveness of the shared memory approach depends on the *latency* incurred on memory accesses as well as the *bandwidth* for information transfer that can be supported. Throughout the thesis, we will formalize and validate a *cost model* to estimate the *memory access latency* when the architecture is subjected to the workload of a parallel application.

2.1 Processing nodes

We focus on processing nodes composed by general purpose CPUs and such that larger-scale systems can be built in a compositional way. The interoperability in the latter systems is achieved by means of apposite interface units that, conventionally, we will indicate with W. The role of W is to intercept and transform memory requests into proper firmware messages, that can be sent either to the interconnection structure (*external* messages) or to some local processing node's units (*internal* messages). Even explicit communication *between processing nodes* can be implemented by means of W.

Notice that a potential re-utilization of uniprocessor architectures for building shared memory architectures is not always free. In fact, specific assembler/firmware mechanisms must be pre-implemented in the uniprocessor itself. Notable examples are synchronization mechanisms (requiring proper assembler instructions or annotations) and cache management for maintaining coherent information among processing nodes. We will treat this topic accurately in a following section.

A processing node can be represented in general as in Figure 2.2. As already mentioned, slight differences between multiprocessors and multi-cores may arise.

- the CPU is a *pipeline* or *super-scalar* uniprocessor (with private data and instructions caches $L1$) exploiting *Instruction Level Parallelism* (ILP), that is parallelism at firmware level. The CPU may exploit

hardware multi-threading, especially in the form of *Simultaneous Multi-Threading* (SMT).

When a shared memory architecture is designed, the choice of the CPU is extremely important. If the sequential performance is a strong pre-requisite, a few yet complex CPUs are used. Otherwise, few large CPUs can be substituted with many simpler CPUs with a gain in efficiency, cost and power consumption; this is also the trend that is characterizing upcoming multi-cores. In turn, the CPU complexity is influenced by the limited chip size. For instance, if there are not hard constraints, a floating point unit could be implemented as a functional unit of the CPU itself. If instead the processing node is part of a multi-core, then problems like chip space and power consumption may arise. In these cases, the solution is to have a set of CPUs sharing a single floating point unit.

- an I/O *Communication Unit* (UC) is provided for explicit interprocessor communication support. As we have already mentioned, though the majority of run-time support information are accessible in shared memory via memory instructions, there are some cases in which direct firmware messages between processing nodes are preferable. Notable examples involve processor synchronization and process low-level scheduling.

- the interface unit W is directly connected to a local memory LM (if present) and some I/O units like UC. As we said, W interfaces the processing node with the rest of the architecture.

- a local memory LM is used for caching information. Caching is notably important for at least two aspects: from one hand it provides, in general, a better instruction service time (local benefit). On the other hand, peculiarly for shared memory architectures, it aims at reducing the shared memory conflicts as well as the interconnection structure congestion, guaranteeing a global performance improvement. LM may be a private processing node's memory (for instance, it realizes the second or the third level of cache hierarchy) or, alternatively, it may be integrating part of the architecture's shared memory (so it can be

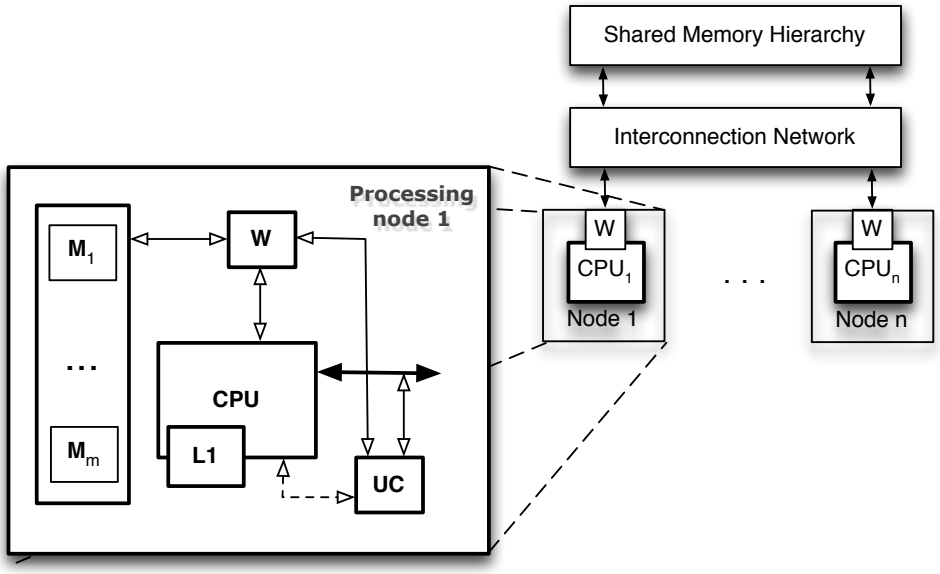


Figure 2.2: Processing Node in a Shared Memory Architecture

addressed by all the other processing nodes). For obvious scalability reasons, the trend in multi-core architectures is to have *small groups* of processing nodes sharing a LM unit.

2.2 Interconnection structures

The task of an interconnection structure (also said interconnection network) in a MIMD machine is to transfer firmware messages from a source node to any desired destination node. Nodes can be processing nodes, memory modules and I/O units. The key parameters to evaluate the performance of an interconnection network are the *bandwidth* and *latency*. High bandwidth and low latency are mandatory to implement scalable MIMD machine. For multi-core architectures, the *cost* of the interconnection network, in terms of both the on-chip area occupied and the energy consumption, is important as well.

There exist a lot of interconnection structures, each one characterized by its own specific bandwidth, latency and cost. In this chapter we do not want

to merely list all of them (the interested reader can consult [8, 20]), rather we will focus on general concepts that will be useful in chapters to come.

Background on interconnection structures Formally, an interconnection structure is a graph $N = (V, E)$, where V is the set of nodes and switches, while E is the set of links between them. The path from a source node to a destination node is called *route* and it is calculated by a *routing algorithm*. It is out of our scope to give complete treatment of routing strategies and algorithms. We only mention that routing can be *deterministic*, i.e. the path is determined solely by its source and destination, or *adaptive*, i.e. the choice of the path is influenced by dynamic events, as traffic intensity, along the way. Further, another important characteristic of a network is how information traverse the route (*switching strategy*). Basically, there are two possibilities: *circuit switching*, i.e. the path between source and destination is established and reserved until it is necessary, or *packet switching*. In the latter, the information is divided into packets. A packet can be individually routed from the source to the destination, since it carries routing and sequencing information in addition to a portion of data. This approach definitely improves the network utilization, since resources remain occupied only for the time needed to forward a packet. Notice that a good shared memory architecture must rely on packet switching networks.

Flow control and wormhole routing The aforementioned routing strategies are directly accomplished by network switches. The other important task of these units is the *flow control*. The flow control aims at solving network contentions by determining when a message can be forwarded along its route. Switches can implement the classical *store-and-forward* technique, or even a more sophisticated strategy called *wormhole* flow control. In the latter, each packet is further subdivided in *flits*. Switches consider flits as an input stream that must be forwarded, without additional buffering, to the same output port. With this approach, packets are still the unit of routing, but the typical benefits of pipeline communications are gained. In this case, we can consider wormhole flow control as an additional source of parallelism. Wormhole flow control has another important property: since flits of the same packet are not buffered before being forwarded, the buffering area

is minimized. Owning to this property, wormhole-based networks become particularly suitable for being used as on-chip interconnection structures in multi-core architectures, thanks to a smaller occupied area.

Network latency When talking about network latency it is important to distinguish between *base* and *under-load* network latency. We define the *base network latency* as the time needed to send a message from a source to a destination node *assuming the absence of network conflicts*. In general, the base network latency depends on many architectural characteristics, e.g average network distance, message length, routing, flow control strategy and so on, but it is not a function of the traffic. Instead, in case conflicts on the network are taken into account, we have the so called *under-load network latency*. In the next sections we show how to evaluate the base latency in wormhole networks.

2.2.1 Base latency in networks with wormhole flow control

According to the methodology of [20], if we consider a message of m words that travels d switches and assuming

- flit size equal to a word
- every unit has clock cycle τ
- every link has transmission latency T_{tr}
- level transaction firmware interfaces

then the network base latency t_{net} is given by

$$t_{net} = (2m + d - 3)(\tau + T_{tr}) \tag{2.1}$$

This formula can be also easily inferred from Figure 2.3.

Notice that pipelined communications in the path for achieving the destination may also occur between units different than network switches. For instance, the W unit inside the processing node or even the memory interface may support the wormhole flow control. In these cases, the expression 2.1 should be carefully instantiated with a proper value of d.

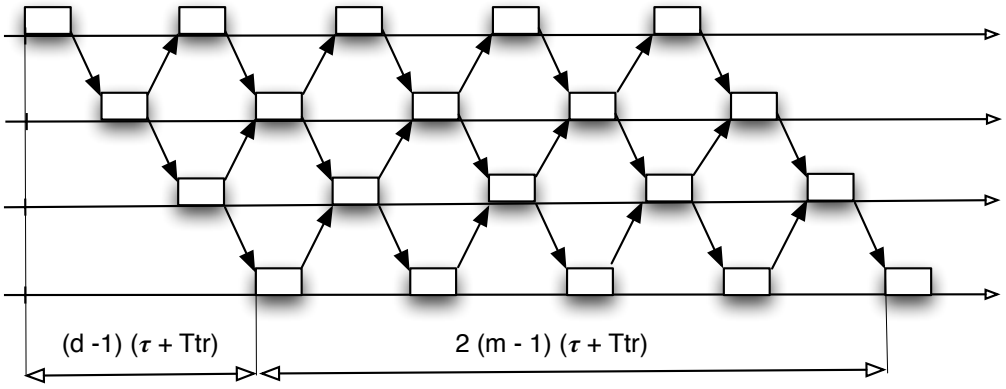


Figure 2.3: Base Latency in pipeline behaviour with level transaction firmware interfaces ($d = 4$ units and message length of $m = 5$ words)

2.2.2 Base latency in time-slot networks on chip

In multi-core architectures the interconnection structure is implemented on chip. The formula 2.1 can be specialized for such architectures by taking into account the following elements. First of all, the transmission latency on chip is negligible, thus $T_{tr} = 0$. Moreover, network switches may exploit time slots-based communication protocols, instead of transaction level ones. In this protocol, switches do not wait for an acknowledgement before sending the subsequent flit of the message. The new latency can be intuitively derived by looking at Figure 2.4. In particular, we have

$$t_{net} = (d + m - 2)\tau \tag{2.2}$$

The expression 2.2 will be useful in Chapter 5 to derive the base latency of a real multi-core architecture. Experiments will show its correctness.

2.2.3 Direct and indirect interconnection structures

Despite interconnection structures will not be first-class citizens of the thesis, for the sake of completeness we analyse and comment the base latency and the bandwidth of the most interesting networks. The analysis is asymptotic with respect to the number n of nodes [20, 8].

Ideally, the best network is the one that offers maximum bandwidth, minimum latency and negligible cost in terms of occupied area and/or power

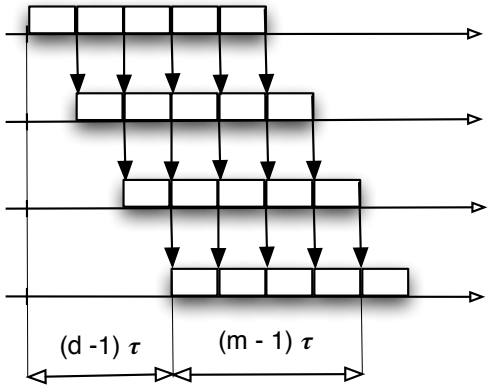


Figure 2.4: Base Latency in pipeline behaviour on chip ($T_{tr} = 0$) and time slot based communication firmware protocols ($d = 4$ units and message length of $m = 5$ words)

consumption. Traditional buses should not be considered as valid interconnection structure for highly parallel machines, since they are not capable of handling simultaneous transfers, i.e. their bandwidth is $O(1)$. On the other hand, fully connected crossbars are not physically realizable when n is relatively large, because of the cost $O(n^2)$. In spite of this, buses and crossbars are actually used, especially in multi-cores when n is very low. Anyway, we remark that if the number of nodes involved is not in the order of few units, other interconnection structures must be chosen.

In the so called *limited degree* networks, a node is either directly connected to a small subset of other nodes, or indirectly connected to every other node by an intermediate path of switches. These interconnection structures can be distinguished for their topology in *direct* or *indirect* networks.

Direct networks In the former case, point-to-point dedicated links connect nodes in some fixed topology. Each node is connected to one and only one switch through the interface unit W. Of course, communication between not adjacent nodes are forwarded by intermediate nodes toward the destination. Notable examples of direct networks are *rings* (2.5 a), *meshes* (2.5 b) and *cubes* (2.5 c).

The base latency for these network is $O(n)$ for a ring and $O(\sqrt{n})$ for a mesh.

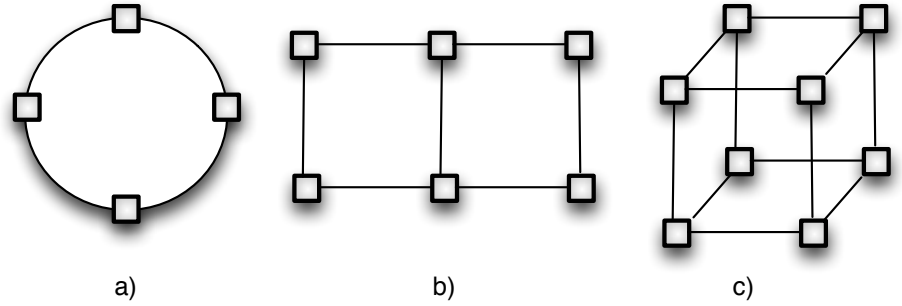


Figure 2.5: Most important Direct Networks with Limited Degree

Indirect networks In indirect networks, nodes are not directly connected as before, but they exploit a sequence of intermediate switches that, in turn, have a limited number of neighbours (i.e. other switches or nodes). In general, more than one switch is used in order to establish a communication between nodes. Notable examples are *trees*, *butterflies* and *fat trees*.

With respect to a tree, the so called fat tree doubles the channel capacity at each level from the leaves to the root, in order to minimize the potential congestion. It is worthwhile to say that all the cited interconnection structures, except the butterfly, connect nodes of the same type. The butterfly, on the other hand, is mainly used to allow communication between nodes of different types (for example n processing nodes with n memory modules). Moreover, it can be used in a very elegant and formal way to implement the so called *Generalized Fat Tree*, an high-bandwidth network (like a fat tree) which is able to interconnect homogeneous and heterogeneous nodes at the same time (like a butterfly).

The base latency in these networks is $O(\log n)$.

Direct and indirect networks for multi-core architectures Nowadays multiprocessors typically exploit cubes or, if the number of processing nodes is high, either fat trees or generalized fat trees are preferred. In multi-core architectures, the area of the chip is an hard constraint that limits many architectural choices, among which the interconnection structure itself. In particular, it is not physically possible to fit complex interconnection structures on-chip, like the class of indirect networks. Therefore, in case an high

number of cores is involved, meshes are used because they are characterized by an acceptable scalability at the price of an easy on-chip realization. A notable example of multi-core architecture using a mesh interconnection structure is the Tilera Tile64, that will be object of study in Chapter 5.

2.3 Shared memory

In a shared memory architecture a single physical address space is shared by all the processing nodes. The problem of performance degradation is a consequence of processing node *conflicts* due to concurrent memory accesses, either referencing private or shared information. Moreover, accessing the memory, processing nodes cause congestion in the interconnection structure as well.

We already know that caching may reduce conflicts on the shared memory, but by itself is not enough. The design of a good memory system is fundamental from the performance point of view. In particular, high bandwidth and minimal contention have to be guaranteed. These goals can be achieved by means of a *modular memory with interleaved organization*. The memory system is then organized in many *macro modules*, with the intent of reducing the number of conflicts by distributing the accesses over the different modules. In turn, a single macro module can be realized either with an interleaved organization or with just one long word-based module. Very often, the number of the internal modules, or the number of words in a long word, coincides with the cache block size, because it allows high bandwidth transfers of cache blocks.

2.3.1 UMA and NUMA architectures

Another important point is the shared memory organization in multiprocessor and multi-core architectures. It is useful to classify these architectures on the basis of such organization.

1. in *Uniform Memory Access* (UMA) the memory modules are equidistant from the processing nodes. This means that the base latency to access them is always the same, independently from both the specific

processing node and the specific macro module. UMA architectures are commonly known also as *Symmetric MultiProcessor* (SMP).

2. in *Non Uniform Memory Access* (NUMA) the symmetry about memory accesses is not more present. If we look at the typical schema of a processing node as illustrated in Figure 2.2, we can consider the shared memory as the union of all the local memories LM of the processing nodes:

$$M = \bigcup_{i=1}^{n} LM_i$$

Hence, LM is not more private of a processing node, but it can be accessed by other "external" processing nodes. Clearly, local memory accesses are much faster than remote ones, since they do not suffer from both the network latency and the potential overhead of memory conflicts (provided that local accesses are prioritized with respect to remote ones).

However, we stress the fact that in both configurations the role of caching is fundamental to reduce the probability of memory conflicts.

In a multiprocessors every processing node has its own interface toward the memory, thus there are not conflicts for accessing it. This does not hold any more for multi-core architectures, since it is not physically realizable to put a memory interface for each core of the chip. This constraint may create an ulterior source of performance degradation.

The distinction between UMA and NUMA shared memory organizations can be also made for multi-cores. If the number of cores is low or a single interface is present, the architecture *resembles* the UMA organization, otherwise it should be considered NUMA.

2.3.2 Base and under-load memory access latency

As soon as a read cache fault occurs, a process P gets stalled for a time t_{a0}, waiting that the requested word is available in cache. In this time interval, a memory request is issued (*request phase*), then it is served by the memory, and finally the cache block is returned to P (*reply phase*). *Assuming the absence of conflicts* (i.e. *unloaded architecture*), t_{a0} is called *base memory*

access latency [20]. To evaluate t_{a0}, we need to consider explicitly the network latency. By resorting on equations 2.4 or 2.5, and adding the latency of those units that do *not* exhibit a pipeline behaviour, we can easily estimate t_{a0}. A practical example on how to evaluate t_{a0} will be provided in Chapter 5.

If we would consider the presence of conflicts, we would have to determine the so called *under-load memory access latency* R_Q. In the following chapter we will see a methodology for deriving R_Q as a function of t_{a0}.

We anticipate that, to evaluate R_Q, a prior study of the *under-load network latency* is necessary; unfortunately this is a quite complex task. However, under certain conditions, the congestion of the network can be assumed negligible. We will study that this assumption is acceptable provided that one of the following condition is verified

1. The interconnection network is a crossbar, a fat tree or at least a generalized fat tree. For these structures we know that the probability of conflict at the network switches is negligible.

2. The interconnection network is realized on-chip (multi-cores) *and* it is *not* a bus. Sophisticated techniques like wormhole flow control *and* time slot based communication protocols must be employed. Further, more than a single memory interface must be provided in such a way that the network traffic can spread among them.

In Chapters 3 and 5 we will validate experimentally the point 2) for the specific case of the Tilera Tile64 multi-core architecture.

2.4 Synchronization mechanisms

In multiprocessor systems, two or more processes could try to modify, *in the same time interval*, a shared data structure D through a sequence of operations S. To guarantee the consistency of D, S must be executed as an *indivisible sequence of operations*. The indivisibility cannot be implemented through the interrupt disabling by itself, as happens in uniprocessor systems. In shared memory systems, processors must be *explicitly synchronized*. Therefore, *locking* mechanisms must be provided at assembler-firmware level.

Two locking primitives *lock* and *unlock* can be realized at assembler level with proper instructions or annotations. They can be used to implement

the mutual exclusion of indivisible sequences, provided that they themselves are *atomic operations*. This last property can be implemented directly at firmware level, resorting on shared memory arbitration mechanisms, i.e. by blocking the access to the memory macro module containing the *locking data structure* (often known as "semaphore"). Obviously, the algorithms of lock and unlock must be simple and fast, in order to release the memory macro module as soon as possible. An efficient solution in terms of memory congestion and fairness, i.e. each CPU is guaranteed to access the lock section in finite time, is the so called *fair locking*. In this solution, the locking semaphore exploits a FIFO queue to record which processors had tried to access the critical section without success. When the unlock is executed, an interprocessor communication is sent to the first outstanding processor (if present), meaning that the critical section can now be accessed.

In multiprocessor systems, the increasing number of processors conflicts due to accesses in mutual exclusion is a well-known phenomena called *software lockout*. This problem can be addressed by using synchronization in a smart way and exclusively at the level of run-time supports. The best approach is to implement the run-time support of concurrency mechanisms such that the *length* of critical sections is minimum, even though their *number* grows [20].

In upcoming multi-cores, synchronization could be implemented without resorting on atomic memory accesses. Indeed, a possibility could be to exploit the on-chip interconnection network to exchange small pieces of information directly between cores. However, this is still matter of study, and no cost models have been provided yet to evaluate the cost of this approach.

Even if we will not study the impact of locking mechanisms in the rest of the thesis, it is important to know that they are needed and thus they introduce some kind of overhead. Most importantly, if locking is implemented and used according to the aforementioned guidelines, then its overhead *can be measured* in the cost model of concurrency mechanisms, as shown in [20].

2.5 Cache coherence

In shared memory architectures, caching is important for both local and global performance improvement. However, shared information must be

maintained consistent among the caches of different processing nodes. This is the so called *cache coherence* problem.

Rather than a formal treatment of the cache coherence problem, which is assumed known (the interested reader can consult [8, 20]), in the following we summarize some aspects concerning the possible solutions. Indeed, in the perspective of formalizing architectural performance models, it is important to understand that different cache coherence strategies have also a different cost in terms of latency and bandwidth [15].

There are two main techniques for addressing the problem.

1. *Automatic cache coherence.* In the majority of the shared memory architectures, there exist firmware mechanisms that automatically guarantee the cache coherence.

2. *Algorithm-dependent cache coherence.* The cache coherence is implemented by the programmer of the run-time support to concurrency mechanisms, without resorting on any primitive firmware mechanism. This is also called software-based cache coherence.

In the former solution, the idea is that every memory writing must be notified to all the CPUs that owns the modified data in their caches. The notification can be done by *update*, i.e. the modified data is sent to all other "sharer" CPUs, or by *invalidation*, i.e. other copies of the shared data become "not valid" as a consequence of an invalidation signalling. Invalidation may seem more complicated, but it has a lower overhead because there is not need to communicate the entire modified information. Mainly, there are two implementation categories to these strategies.

1. *Snoopy-based* implementations use a centralization point at firmware level to notify modified information or invalidation messages. Typically, these techniques are used when the interconnection structure is a bus or a ring.

2. a *Directory-based* approach is useful when the number of CPUs increases and more complex interconnection structures are used. The idea is to use some data structures to record which cache contains a certain block. This approach reduces the overhead but it has a cost,

i.e. congestion comes out in spite of the directory should be allocated in a fast memory.

Directory-based strategies can be also realized following an algorithm-dependent approach. The idea is to design an explicit management of cache coherence in the run-time support code to the concurrency mechanism. In theory, this approach does not introduce inefficiency per se, but software lockout impact could be aggravated.

Automatic cache coherence is implemented in the majority of shared memory architectures, without providing the possibility of turning it off (to allow, for instance, the implementation of an algorithm-based approach). Actually, this trend seems to change in light of the fact that a few upcoming multi-cores provides the possibility of disabling automatic cache coherence; it is the case of the Tilera Tile64, the multi-core architecture object of study in Chapter 5.

2.6 Structured programming on shared memory architectures

Methodology We advocate that parallel applications should be written according to a structured methodology, without being influenced by the characteristics of the specific architecture. In *structured parallel programming* [20] (or *skeleton* based parallel programming) a limited set of *parallel paradigms* (skeletons) is used to express the structure the parallel application. One of the common critics to this approach is that the freedom of the programmer is limited. Actually, properties like *simplicity* and *composability*, as well as *parametric cost models*, make structured parallel programming a very attractive approach to dominate the complexity inherent the design of parallel and distributed application.

Ideally, a compiler should exploit the cost model of a parallel paradigm to *evaluate* and *optimize* certain performance parameters of the application, like processing bandwidth, completion time, latency, efficiency, scalability and so on. The cost model is characterized by a set of functions depending in large part on two parameters:

- the calculation time of the sequential algorithm, T_{calc}

- the interprocess communication latency, T_{send} (assuming to work in a message-passing environment). It measures the latency for completing an interprocess communication, i.e. to copy the message into the target variable and to perform all the needed run-time support actions.

T_{send} represents a very strong abstraction of the concrete architecture: all the aspects we treated in the previous sections, like UMA vs NUMA memory organization, interconnection structures, synchronization, cache coherence techniques and so on, are captured in this fundamental parameter.

An *abstract architecture* captures the essential characteristics and features of several, different physical architectures. The specificity of each individual physical architecture is expressed by the *cost model of the abstract architecture*, which is represented by T_{send} and T_{calc}. A programmer, or even better a compiler, "reasons" on the simpler abstract architecture, instead of the concrete one, and "uses" its cost model, for instance, to instantiate the parallel paradigm's cost model. To avoid confusion, we remark that there are two cost models

- An "application" cost model relative to the parallel paradigms adopted for structuring the computation, which are architecture independent.
- An "architectural" cost model associated to the abstract architecture, expressing parameters like T_{send} and T_{calc}.

Anyway, for being instantiated, both cost models require parameters of the specific parallel application.

Cost models should be exploited at both compile- and loading-time to perform those optimizations that nowadays are done for sequential code. This approach should simplify the *development* of parallel applications (e.g. by using high-level and user-friendly tools) and should solve the performance *portability* and *predictability* problems as well. For instance, besides introducing platform-specific optimizations, a compiler could optimize the structure of the parallel computation itself. Moreover, if more than one implementation of an application were provided (to exploit certain peculiarities of a concrete architecture), then the compiler could choose the most effective one for a specific target architecture. Other important properties, like adaptivity and dinamicity, could be guaranteed as well.

The gap between hardware and software The gap between structured parallel programming and shared memory architectures is still wide. Despite the theory behind structured parallel programming is solid, currently there is no way to *predict* the performance of a program at different parallelism degrees or on different architectures. In other words, the development of a solid architectural cost model is still incomplete, although some theoretical work already exists [20]. Without such cost model, the methodology of the previous paragraph cannot be applied efficiently.

Developing a general architectural cost model is a rather complex task. The difficulty is to derive, for each architecture, good approximations of parameters like T_{send} and T_{calc}. The complexity comes from the considerations we made in the previous sections: the sharing of units, especially the memory hierarchy, surely optimize the global architecture performance, but at the same time makes single flow performance less predictable. For instance, in Section 2.3 we understood the problems behind the prediction of the memory access latency R_Q. If we were able to estimate R_Q, then we would make a fundamental step toward the definition of the architectural cost model, *because parameters like T_{send} and T_{calc} can be expressed as functions of R_Q* [20].

The objective of the thesis falls in this area. We want to extend the work of [20] and validate it against experimental results. The final result will be the formalization of a cost model to determine meaningful approximations of the *under-load memory access latency* in shared memory architectures. Particular care will be reserved for state-of-the-art multi-cores.

Chapter 3

Queueing theory-based cost models

In the previous chapter we have understood the importance of defining an abstract representation of a concrete architecture. This abstract architecture must be accompanied by a cost model. A cost model is fundamental to estimate the parallel application performance by taking into account both the features of the application itself, the concrete architecture and the structure of the run-time support to concurrency mechanisms. We strongly advocate that a cost model should be easy to use and conceptually simple to understand. In this perspective, we have shown the idea of capturing all architectural and run-time support aspects in two simple functions T_{send} and T_{calc}. The knowledge of these function would be of invaluable importance for a programmer or (even better) a compiler to evaluate, configure and optimize parallel programs. Unfortunately, as we have already seen, shared memory architectures are heterogeneous, extremely complex systems; this fact, together with the inherent complexity of parallel programs, makes it really hard the derivation of the abstract architecture's cost model, that is a good approximation for T_{send} and T_{calc}. For instance, a critical problem is to predict in which measure the limited memory bandwidth will influence the value of these parameters. To answer this question we have to estimate the so called *under-load memory access latency* R_Q, that is the average time to access the main memory subjected to the workload of a parallel application. T_{send} and T_{calc} will be expressed as functions of R_Q. As far as we know, apart from [20], there are not any studies in literature addressing this topic with

our methodology.

The structure of this chapter is as follows:

1. Firstly, we will introduce some Queueing Theory concepts, since the cost model will be based on them.

2. Secondly, we will formalize a general methodology to estimate R_Q in shared memory architectures. The key idea is very simple: mapping the system architecture on a Queueing Network. We will see the weaknesses of this approach that will force use to look for another approach.

3. Then, we will describe a second methodology based on a simpler architectural model [20]. The main feature of this approach will reside in the simple analytical resolution technique.

4. As our contribution, we will improve the latter model by showing a new resolution technique.

5. Finally, we will compare and validate the model resolution techniques of points 3) and 4) against experimental results.

3.1 Elements of Queueing Theory

We will formalize a cost model basing on Queueing Theory concepts. Thus in this section we will refresh and summarize important results regarding both simple and intermediate queueing systems; the reader may consult [14, 4] for a deeper understanding of those concepts that here will be just reviewed.

3.1.1 Description and characterization of a queue

Description of queues A queueing system models the behaviour of a server S where clients (often known as jobs or client requests) arrive and ask for a service. In general, clients have to spend some time in a queue Q waiting that S is ready to serve them. The scheme in Figure 3.1 is a *logical* one, not necessarily corresponding to the real structure of the system we are modeling. For instance Q could not physically exists or it could be even distributed among the clients. However in some of these cases it turns

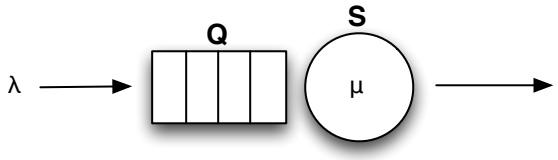


Figure 3.1: A queue

out to be easier to study the whole system as a single logical queue. This kind of approximation can drastically reduce the complexity of the analysis and makes it possible to obtain an approximate evaluation, which is however meaningful provided that the mathematical and stochastic assumptions are validated. We will use and explain this approach in the next sections.

Queue models are classified according to the following characteristics.

- The stochastic process A that describes the arrivals of clients. In particular, we are interested in the probability distribution of the random variable t_A extracted by A. t_A represents the *interarrival time*, that is the time interval between two consecutive arrivals of clients. Its mean value is denoted by T_A, the standard deviation by σ_A and the mean rate of interarrivals by $\lambda = \frac{1}{T_A}$.

- The stochastic process B that describes the service of S. B generates the random variable t_S that represents the *service time* of S, that is the time interval between the beginning of the executions on two consecutive requests. Its mean value is denoted by T_S, the standard deviation by σ_S and the mean rate of services by $\mu = \frac{1}{T_S}$.

- The *number of servers or channels* r of S, that is the parallelism degree of S. In the following, except for some specific cases, we will assume $r = 1$, that is a sequential server.

- The *queue size* d, that is the number of positions available in Q for storing the requests. Notice that in computer systems this size is necessarily fixed or limited. Unfortunately most of the results in Queueing Theory have been derived for infinite length queues. However, the results provided for infinite queues will sufficiently approximate the case of finite ones, under assumptions that we will discuss case by case.

- The *population* e of the system, which can be either infinite or finite.
- The *service discipline* x, that is the rule that specifies which of the queued requests will be served next. We will use the classical $FIFO$ discipline.

Basing upon these information, queues can be classified according to the standard Kendall's notation (see [14] for more details). For instance, we will indicate with $M/M/1$ the queue with a single server where both the input and the service processes are Poisson ones.

Interdepartures process The stochastic process C that represents the departures from the system (interdeparture process) is dependent on the nature of the queue. For $A/B/1$ queues, being T_P the average interdeparture time, an evident result is that $T_P = max(T_A, T_S)$.

A first interesting property is the following (see [20] for a simple proof):

Theorem 1 ***Aggregate inter-arrival time.*** *If a queue Q has multiple sources (i.e. multiple arrival flows) each one with an average interdeparture time T_{p_i}, the total average interarrival time to Q is given by:*

$$T_A = \frac{1}{\sum_{i=1}^{N} \frac{1}{T_{p_i}}}$$

Characterization of queues A first average measure of the *traffic intensity* at a queue is expressed through the *utilization factor* ρ.

$$\rho = \frac{\lambda}{\mu} = \frac{T_S}{T_A}$$

For our purposes an extremely important situation is given by $\rho < 1$. Under this situation the system *stabilizes*, therefore it becomes possible to determine the so called *steady-state* behaviour of the system.

Other metrics of interest to evaluate the performance of a queueing system are:

- the *mean number of requests in the system*, N_Q: the average number of client requests in the system including the one being served;

- the *waiting time distribution*: the time spent by a request in the waiting queue. We are practically interested in its mean value W_Q.
- the *response time distribution*: with respect to the waiting time distribution, it includes also the time spent in the service phase. We will denote its mean value as R_Q. Notice that $R_Q = W_Q + L_S$, where L_S is the average service latency .

A very general result that can be applied to different kind of scenarios (not just Queueing Theory) is the Little's theorem.

Theorem 2 ***Little's law.*** *Given a stable system ($\rho < 1$) where clients arrive with a rate λ and the mean number of clients in the system N_Q is finite, the average time spent by a client in the system R_Q is equal to*

$$R_Q = \frac{N_Q}{\lambda}$$

The reasoning behind this theorem is intuitive, while the proof is quite complicated. The interested reader may consult [14] for a deeper explanation.

3.1.2 Notably important queues

The Queueing Theory is extensive and treats an incredible large number of special queues (that is, queues with a specific configuration $A/B/r/d/e/x$), some of which also particularly complicated. In order to keep limited the complexity of deriving the architecture cost model, we will be interested in a minimal (yet meaningful) subset of these queues. Therefore in this section we illustrate the main results for only two peculiarly configurations: the $M/M/1$ and the $M/G/1$ queues.

The $M/M/1$ queue In a $M/M/1$ queue the arrivals occur according to a Poisson process with parameter λ. The services are exponentially distributed too, with rate μ. The memoryless property of the exponential distribution, besides being simple to model, is very important in our context because it allows us to approximate a lot of different meaningful scenarios. The service discipline is FIFO and it is assumed that the queue size is infinite. It can be shown that the average number of requests in the system is equal to

$$N_Q = \frac{\rho}{1 - \rho}$$

Applying the Little's law we obtain:

$$W_Q = \frac{\rho}{\mu(1-\rho)}$$

$$R_Q = \frac{1}{\mu(1-\rho)}$$

It could be also proved that even if the queue is of finite size k, the previous formulas still represent an acceptable result provided that the probability that a request gets stuck due to the full queue is an event with negligible probability. We will show that in our models we will be always able to work under this condition.

The $M/G/1$ queue Although very common, the hypothesis on the exponential distribution of the service time could not be applicable in some concrete case of interests. For instance, there could be architectures in which the memory subsystem takes a *constant* amount of time to handle a processor request. In these cases we are interested in the deterministic distribution. In the following chapters we will see other cases in which we may not use the exponential probability distribution.

We introduce the $M/G/1$ queue, where the symbol G stands for *general* distribution. All assumptions and considerations made for the $M/M/1$ are still valid, except for the distribution of the services: indeed with an $M/G/1$ we are able to model any distribution of the service time. For this queue we get the following fundamental results (coming from the so called *Pollaczek-Khinchine formulas*):

$$N_Q = \frac{\rho}{1-\rho}\,[1 - \frac{\rho}{2}(1-\mu^2\sigma_S^2)]$$

Applying the Little's law:

$$R_Q = \frac{1}{\mu(1-\rho)}\,[1 - \frac{\rho}{2}(1-\mu^2\sigma_S^2)]$$

As we said, a particular case of interest is the one of $M/D/1$ queue where the service time distribution is *deterministic*, that is the variance is null. Imposing $\sigma_S = 0$ in the previous formula we get the expression of the average response time for a $M/D/1$ queue.

3.1.3 Networks of queues

Queueing networks in general A queueing network is a system where a set of queues are interconnected in an arbitrary way. The state of the system is typically represented as the number of jobs currently occupying each queue. Figure 3.2 shows the simplest queueing network, that is two $M/M/1$ queues connected in series.

Figure 3.2: Two $./M/1$ queues in series.

Actually, the arrival process at the latter queue is exactly the output process of the former one; thus it is more correct to identify the second queue with the notation $./M/1$. This expresses the fact that the arrival process at the second queue is dependent from the rest of the network.

There exist different classes of queueing networks. A first distinction can be made among cyclic and acyclic networks. It is also useful to distinguish between open, closed and mixed networks. The classification is particularly useful because several theorems show that, for specific classes of networks, there exists the possibility of deriving a so called *product-form* solution. Solving a queueing network in product-form means that the performance of the whole system can be analytically derived in a compositional way, starting from the analysis of single queues in isolation. The key point is that a lot of different algorithms exist to evaluate the performance of product-form networks. This means that if we were able to model an architecture as a product-form queueing network, then we could apply an algorithm to extract some parameters of interest, like the system waiting time, and use them to estimate the under-load memory access latency. Unfortunately, we will see that things are not so simple.

Closed queueing networks In a closed queueing network there cannot be neither arrivals nor departure outside the network. Thus the population of the network is constant. Equivalently, for reasons that will be clear in the next section, we like to think at these networks as systems where *a new*

request is allowed to flow only when another request departs from the network. Therefore, this kind of networks are particularly useful for modeling systems with *finite capacity.* Figure 3.3 shows the simplest closed queueing network.

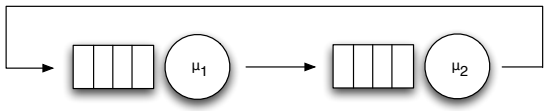


Figure 3.3: A closed system: two $./M/1$ queues in series with cycle.

BCMP Networks We end up this overview by showing one of the main results of Queueing Theory, that is the BCMP Theorem, which will be useful in the next section. This theorem defines a wide class of networks characterized by a product-form solution. The possible features of a BCMP network are the following:

- *Classes of clients.* The BCMP theorem introduces the concept of *class.* A class is a set of clients that share the same *routing matrix.* For a client c of class r, the routing matrix expresses the probabilities $p_{r,i,j}$ that c, once serviced at the queue i, goes to the queue j. This implies that clients belonging to different classes can behave differently for what concerns the paths inside the network.

- *Service disciplines* at a queue can be different from the classical FIFO (e.g. LCFS).

- *Service time distributions* can be different than the classical exponential. In some cases, load-dependent service times are allowed (i.e. service time dependent on the number of clients currently in the queue).

For some specific combinations of service disciplines, service distributions and number of servers at a queue, the BCMP theorem claims that a product-form solution can be derived. For purposes that will be clarified in the next section, we are just interested in the following corollary.

Theorem 3 ***BCMP Networks.*** *Consider a closed queueing network in which clients can belong to different classes. Assume that all queues of the network are characterized by:*

- *a single server (sequential server);*
- *a FIFO service discipline;*
- *exponential service time;*

then for this kind of networks a product-form solution exists.

Finally, notice that claiming that a product-form solution exists, does not imply that it is also "simple" to determine it. For instance, the time- and space-complexity of some resolution algorithms could be polynomial in the number of queues, but exponential in the number of classes, and vice-versa. For certain networks, some algorithms could also suffer from numerical instability.

3.2 Processors-memory system as closed queueing network

At the beginning of the chapter we pointed out the necessity of estimating the under-load memory access latency R_Q. Knowing this parameter is fundamental to express the cost model for an abstract architecture. In this section we show the most intuitive way to model a multiprocessor system, that is mapping it on a queueing network. Basing upon this model we will explain how, in principle, we could determine R_Q. In spite of the apparent simplicity, we will early understand that this methodology hides a lot of subtle problems.

3.2.1 Formalization of the model

Consider a shared memory system in which each of the n processing node is connected, through some kind of interconnection network, to all memory modules (or equivalently, to the chip's memory interface unit in case of a multi-core). A parallel application composed of n processes is being executed. The process computation alternates *think* to *wait* periods. During a *think* period a process P is working on registers or data stored in its local cache. At some point a cache fault occurs and a block request is issued to the main memory. P stops working until the memory request is satisfied, i.e. until

the requested block is sent back to the P's cache. The duration of this latter *wait* period, that can be strongly influenced by the workload generated by the other $n-1$ processes of the parallel application, corresponds to R_Q.

We can model this system as a closed queueing network, like in Figure 3.4.

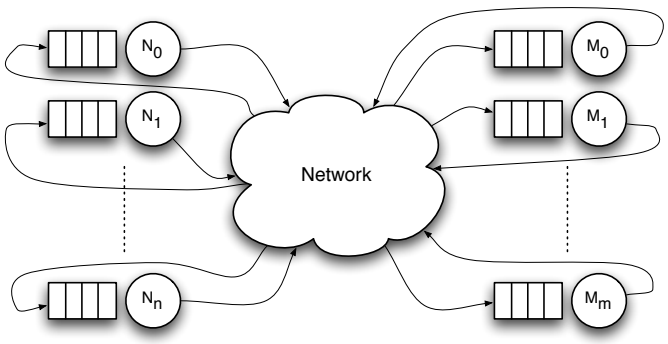


Figure 3.4: A closed queueing network model for a shared memory architecture.

We identify:

- processing nodes (N), memory modules (M), interface units and other firmware units (e.g.: network routers) as queues.
- the memory requests/replies as the unit of flow of the network; in other words, they are the *clients* (jobs) of the queueing network.

In the queueing network there are n clients, exactly one for each process. This is because we assume that each process can issue only a memory request at a time. During a *think* period, a client r resides at the processing node queue N_i. Once the *think* period expires, r departs from N_i and gets routed, through the interconnection network, toward the memory module queue M_j. In this phase, r represents a memory request that has been issued after a cache fault. Once serviced at M_j, r is routed back to P. In this second phase, r models the reply of the memory system.

By looking at Figure 3.4, it is clear that each client is characterized by its own path. For instance, a request generated by N_i will travel across different queues with respect to a request generated by N_j, $i \neq j$. To implement this aspect in the queueing network, we apply the concept of classes of clients.

Assume also that

- The service time at each queue is exponentially distributed; this is a fairly acceptable approximation for the processing nodes, while some problems arise for what concerns the memory system, as we will see in the next sections.

- The queues are characterized by sequential server, that serves requests in a classical FIFO manner. This is in general a meaningful assumption.

Then, in light of this model, we can straightforwardly apply the BCMP Corollary (Theorem 3). This means that the queueing network admits a product-form solution, and an algorithm can be used to extract metrics of interest, for instance the average response time R_{Q_i} at each queue i of the network.

Let $Path$ be the multiset of queues that have to be traversed by r to go from N_i to M_j and vice-versa, with $M_j \in Path$ and $N_i \notin Path$. The performance index we are interested in is the average times R_{Q_i} spent by r at each queue $i \in Path$. We can estimate the under-load memory access latency R_Q as:

$$R_Q = \sum_{i \in Path} R_{Q_i}$$

3.2.2 Performance analysis of the model

Solving the closed queueing network model of a shared memory architecture is the process of determining R_Q. First of all, we need to parametrize the model, i.e. we have to fix some values of the queueing network, among which the service time at each server. We notice that:

- the service time at nodes N_i $(i = 1...n)$ corresponds to the process *think* period. Its average value T_P is a parameter of the *sequential algorithm*, thus it can be easily derived by profiling.

- the service time at memory modules M_i $(i = 1...m)$, with mean value T_S, is an *architecture-dependent* parameter, thus it is known in advance.

There are n classes, one for each process. Each class is associated its own routing matrix M_i: this way it is possible to route a request to a certain

memory module and, at the same time, sending back the answer to the processing node that originated it.

At this point, we need an algorithm that takes as input data these information and produce as output the performance metrics of the queueing network, e.g. throughput and waiting time at each server. The best algorithm to solve this kind of product-form networks is the *Mean Value Analysis* (MVA) algorithm [18, 12].

MVA allows us to compute average queue lengths and response times, as well as throughputs. MVA is a conceptually simple algorithm based on two important theorems: the *Arrival Theorem* [18] and the *Little's law*. The time- and space-complexity of MVA is polynomial in systems with a single class of clients ($O(n^2)$), while it grows exponentially with the number of classes. Since our model is a multi-class one, we could either:

1. *simplify* and *modify* our model by using a single class of clients (even changing drastically it),

2. or use different versions of the original algorithm, that go under the name of *Approximate Mean Value Analysis* techniques. These algorithms find out *approximations* of the expected solution, mitigating the problem of exponential time complexity [12].

At first sight we could be tempted to opt for the second solution. Exploiting a well-known algorithm to solve the model, although obtaining only an approximated solution, is an inviting perspective. However, we need to be care of the following aspects.

- **Complexity of the actual model.** Building the closed queueing network model of a shared memory system is not so straightforward. Figure 3.4 is just a logical scheme: it suffers from the lack of the network model, the shared memory hierarchy, the potential parallelism within the processing node and so on. Clearly, representing all these elements in our model would be nonsense because of the exceeding complexity. Therefore we need a trade-off. We advocate that at least the memory hierarchy, when shared by a set of processors, should be modeled.

- **Importance of qualitative reasoning.** We claim that a cost model, to work, must be simple. Necessarily simple to understand, ideally sim-

ple to evaluate. The architectural model we have discussed earlier is neither of them. It is not simple to study and, moreover, it is not possible to intuitively foresee how a change in the parallel application will be reflected on the final performance, at least until a new instance of the MVA algorithm will be executed. We would like an analytical model, e.g. some kind of simple equations, that help us in understanding, for instance, how R_Q varies as a function of T_P or T_S. Unfortunately, it is extremely complex to derive such equations from the actual model, even with a deeper knowledge of the Queueing Theory.

- **Flexibility of the model.** The exponential distribution is often a good approximation for our purposes, but not always. There can be cases in which a server is in reality a deterministic one, e.g. a memory module that takes a constant time to retrieve the desired information. The problem is that if we release the assumption of exponential service time, the $BCMP$ theorem (3) does not hold any more. In general, networks with servers having service times different than the exponential one cannot be reduced in product-form. In this case, the stochastic modeling of the system is no more Markovian. This is perhaps one of the biggest problem in performance modeling, and not only in our context. There exist approximated versions of MVA, based on heuristics, that try to solve this limitations, but results are often not as good as expected. Nevertheless, our experience suggests that it is very common to encounter new scenarios in which MVA either is not sufficient to solve our model (because its assumptions are violated) or requires a partial redesign to accommodate our necessities.

In light of these considerations, instead of studying MVA techniques for a complex architectural model, we prefer to *simplify* it and looking for new, easier ways of computing its performance measures.

In the next section we will show a simplified version of this model. However, some of the concepts that we have introduced will be exploited as well in the following.

3.3 Processors-memory system as client-server model with request-reply behaviour

3.3.1 Formalization of the model

Consider a system in which a set of N client modules $C_1, C_2, ..., C_N$ send requests to a server module S and need to wait for an explicit reply in order to continue their elaboration. An example of this scheme is shown in Figure 3.5. Notably cases of this interaction pattern are some client-server parallel applications as well as processors-memory systems. Therefore, the model formalized in this chapter may be applied to a lot of different domains. The main goal in a client-server system with request-reply behaviour is to estimate the average response time R_Q of S.

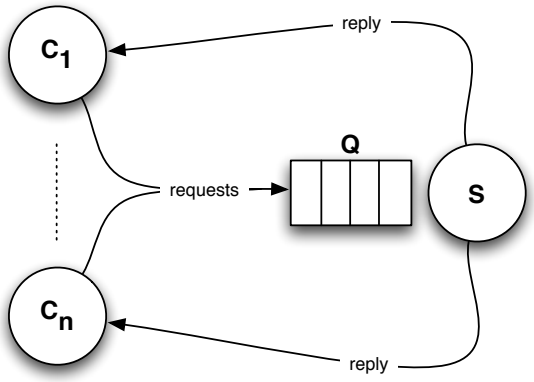


Figure 3.5: Client-server system with request-reply behaviour.

A *logical* queue Q is present in front of S. We talk about a logical queue because conflicts for resource contention could happen not only in S, but also nearby other modules that, for complexity reasons, are abstracted away from the system. For example, think to a scenario in which client's messages need to travel along an interconnection network to reach S; it is unlikely that the network is a crossbar, thus the probability $p_{conflict}$ that a request has to be queued somewhere in the network is different than 0. Depending on the value of $p_{conflict}$, the cost model can be properly parametrized to take care of such conflicts. For example, a very simple yet meaningful approach consists in increasing the service time T_S of S. In these cases we may say that S is

logically the subsystem that includes both the interconnection network and the memory module that carries out clients' requests.

We instantiate the model on a generic multiprocessor system with N processing nodes and m shared memory macro-module by using the same methodology of [20].

- The processing nodes become the clients $C_1, C_2, ..., C_p$ of the system.
- We assume for simplicity that $C_1, C_2, ..., C_p$ have an identical behaviour. The behaviour is the one described in the previous chapter, where *think* periods alternate to *wait* ones. The duration of a *think* period is represented by an exponentially distributed random variable, with mean value T_P.
- S is the shared memory macro-module *and potentially* even the interconnection network paths from the p nodes to the memory macro-module itself.
- Let p be *the average number of processing nodes sharing the same memory macro-module*. It is very important that p is as low as possible in order to minimize the congestion overhead at a memory macro-module. In an SMP architecture, in which statistically the memory accesses are uniformly distributed over the m macro-modules, p can be estimated as the mean of the binomial distribution, i.e. $p = \frac{N}{m}$. In a NUMA architecture, the uniform distribution does not hold any more; here, the value of p is dependent on specific characteristics of the parallel program. It has been shown [20] that, for structured parallel programming, there exist optimum strategies and heuristics to map processes onto processing nodes, in such a way to minimize the value of p. We will see an example of these "smart" mappings in the next chapters;

3.3.2 Assumptions and variants

The cost model for determining R_Q implies, in general, a complex evaluation due to the large number of degrees of freedom. We have already seen that a lot of problems arise when trying to model the architecture as a general closed queueing network. With the client-server approach we remarkably alleviate the complexity:

1. *by simplifying the original model.* The complex closed queueing network shrinks to a single queue model. Processing nodes become simple modules that generate requests with a certain frequency. The focus is on a single memory macro-module rather than the whole memory system. The network model is cut away (network conflicts overhead may be taken into account during the resolution of the model).

2. *by using a simple, yet meaningful, analytical resolution technique* that we will study in the next section.

Since we are seeking for a resolution technique characterized by reasonable complexity and, at the same time, that is able to retrieve approximated results, we need to rely on some further assumptions and simplifications (some of them will be released in the next sections).

- We have already said that T_P is the mean value of an exponentially distributed random variable. Actually, this distribution depends on the parallel application characteristics, and could be even different from the exponential one. For instance, when the elements of an array are read linearly and the computation between one read and the subsequent takes always the same amount of clock cycles (which is quite common in a program), the proper distribution should be the deterministic one. However, we are rather interested in evaluating the interarrival time at S. Since we are assuming independent processing nodes, the input stochastic process at S is assumed to be a Poisson one, with constant rate $\lambda = \frac{1}{T_P}$. Consequently, Q will be modeled as a special $M/B/1$ queue, with $B \in \{\text{M, D}\}$. In reality, as we will see in Section 3.3.4, this approximation can be even further accurate.

- We focus on the server service time T_S. The behaviour of a memory macro-module is, in most of the cases, deterministic [20]. That is, any request generated by a processing node takes always the same amount of clock cycles to be served. On the other hand, there are also memory systems that exhibit a non-trivial behaviour. For instance, some kind of memories can exploit the space- and time-locality of groups of consecutive requests for elements stored on the same row of a memory

bank [21]. In these cases, T_S is not constant any more and an exponential distribution could be used as a better approximation. Other memories (e.g. DDR-2 memories) have a *load-dependent* behaviour: the more the number of request in Q, the lower is the average service time. This is because requests can be reordered in such a way to exploit the aforementioned locality properties. In Chapter 5 we will study a real architecture characterized by a memory load-dependent system. In this chapter, we will focus only on exponential and deterministic service times.

- We are assuming homogeneous clients. However, processes of a parallel application can exhibit different behaviours each other, especially in case of structured parallel programming, where different parallel paradigms (i.e. skeletons) can be used within the same application. Moreover, the general structure of the application itself is abstracted away in the model. For instance, the presence of specific communication patterns (i.e. stencils, see [20]) could be considered.

- We are modeling non-hierarchical systems, i.e. architecture where only the main memory is shared. Instead, especially in state of the art and upcoming multi-cores, the trend is to provide cores with shared levels of caches. The intuition is that concurrent accesses to shared resources can introduce a significant overhead, especially if the number of sharers is large. The problem of hierarchical architectures is addressed by [5].

3.3.3 Model resolution

In [20], the following system of equations is proposed as resolution technique of the client-server system.

$$\begin{cases} T_{cl} = T_P + R_Q \\ R_Q = W_Q(T_s, \rho) + t_{a0} \\ \rho = \frac{T_S}{T_A} \\ T_A = \frac{T_{cl}}{p} \\ \rho < 1 \end{cases} \tag{3.1}$$

Each client generates the next request only when the result of the previous one has been received. The behaviour of a client, as we have already said, is cyclic: *think* periods (T_P) alternates to *wait* ones (R_Q), leading to a certain client average interdeparture time T_{cl}. This fact is captured by the equation $T_{cl} = T_P + R_Q$. Once we know T_{cl}, we can determine the server average interarrival time T_A; by resorting on Theorem 1, we have that $T_A = \frac{T_{cl}}{p}$. The utilization factor of the system is given by $\rho = \frac{T_S}{T_A}$. Finally, the underload memory access latency R_Q is simply given by the average waiting time W_Q plus a constant known in advance, which is the base latency t_{a0} (see Section 2.3.2). The expression of W_Q depends on the type of Q. For instance, for a $M/D/1$ queue we have $W_Q = T_S \frac{\rho}{2(1-\rho)}$.

The system has a self-stabilizing behaviour: e.g. a temporary increase of T_A has the effect of decreasing R_Q, that in turn tends to lower T_A itself since T_{cl} will decrease. This is also an example of qualitative reasoning. Since the system shows a self-stabilizing behaviour, it could be proved through markovian analysis that $\rho < 1$. This means that a steady-state solution exists.

Assuming that Q is either $M/M/1$ or $M/D/1$, solving this system with respect to R_Q leads to a second degree equation in ρ. The two solutions ρ_1 and ρ_2 are always such that $\rho_1 < 1$ and $\rho_2 > 1$, thus the solution of the model must be subjected to the constrain $\rho < 1$.

Although suffering from the limitations of the previous section, this model resolution technique is very interesting because:

- it is simple;
- it is based on mean values quantities rather than probability density

functions, and this further simplifies the analysis. We advocate that a resolution technique based on mean values is sufficiently accurate for our purposes.

- it enables qualitative analysis;
- it is good also for quantitative analysis (i.e. it gives quite good approximation to the real value of R_Q, as we will see in Section 3.3.5);
- it is parametric in the service times distribution. The formula of the average waiting time W_Q is chosen according to the scenario we are modeling. For example, if the memory subsystem shows a deterministic behaviour, than we will use the standard Queueing Theory formula for $M/D/1$ queues (3.1.2).

Besides the distribution of the server service time, another important aspect is the choice of the parameter of this distribution, that represents the frequency $\mu = \frac{1}{T_S}$ at which requests are carried out. Consider the two extreme cases, which are also the most important ones:

- $T_S = t_{a0}$. The server service time is the base latency. In this case we model the system as if the network between clients and server would be a bus. It is known that a bus can handle only one request at a time; this behaviour would be captured by our system, since client requests would be blocked immediately in the processing node.
- $T_S = T_M$, being T_M the average time required by a memory macromodule to carry out a request. This is the case wherein network conflicts are neglected. This assumption is meaningful in particular types of networks, e.g. crossbar or fat-trees. We claim that this case is particularly interesting also when modeling multi-core architectures (unless the network is a bus) because the time needed by a request to be routed within the chip's network is significantly lower than the one spent nearby the memory (queueing delay plus service time).

In the following, we will assume $T_S = T_M$.

Finally, remember the original goal: we are determining the cost model of an abstract architecture. The abstract architecture is characterized by two

functions: T_{send} and T_{calc}. To express these functions, we had to understand the system ability to execute a certain amount of instructions *in presence of memory conflicts*, that is the real bandwidth of processing nodes. In this perspective, the value of R_Q (or T_{cl}) will be used to express such functions, as shown in [20].

3.3.4 A new resolution technique for exponential servers

In this section we propose an analytical resolution technique that removes an approximation of the previous model. The intuition is the following. Each client generates the next request only when the result of the previous one has been received. This means, from a model point of view, that it is equivalent to consider a *constant population of jobs*, as many as the total number of clients p, that circulate endlessly and never leave the system. Since the population is fixed, the rate of arrivals at S cannot be constant, otherwise it would be in contrast with this observation. Rather, the frequency of arrivals will be *proportional* to the number of clients that are neither queued nor in service. Being i the number of clients in a *think* phase, we have an arrival frequency equal to

$$\lambda_i = i\lambda, \quad 0 \leq i \leq p \tag{3.2}$$

Consider a queueing system wherein the population consists of p clients. The server service time is exponentially distributed with mean value T_S. T_P is the mean value of an exponential random variable representing the duration of the client's *think* phase. We notice that

$$T_P = \frac{1}{\lambda} + T_{net}$$

T_{net} is a constant of the system, representing the network base latency to reach the server (see Section 2.2). For such systems, the probability π_k of having k clients ($k = 1, 2, ..., p$) inside the queue is equal to [4]:

$$\pi_k = \pi_0 \ (\frac{T_S}{T_P})^k \ [p(p-1)...(p-k+1)] = \pi_0 \ (\frac{T_S}{T_P})^k \ \frac{p!}{(p-k)!} \tag{3.3}$$

$$\pi_0 = (1 + \sum_{j=1}^{p} (\frac{T_S}{T_P})^j \ \frac{p!}{(p-j)!})^{-1} \tag{3.4}$$

We notice that π_0 is the probability that the server is idle, i.e. there are no requests that are being served. Therefore, the system utilization factor is given by:

$$\rho = 1 - \pi_0 \tag{3.5}$$

As usual, $\rho < 1$ means that the system self-stabilizes and the steady-state evaluation is meaningful. Hence, we can impose:

$$\frac{T_S}{T_A} = \rho$$

Thus, we obtain

$$\frac{T_S}{T_A} = 1 - \pi_0$$

From Theorem 1, we know that $T_A = \frac{T_P + R_Q}{p}$. Substituting, we finally end up with:

$$R_Q = \frac{p\ Ts}{1 - \pi_0} - T_P \tag{3.6}$$

Notice that this analysis is based upon the steady-state condition of a Markov chain, while no result regarding a specific queue type has been used. Thus, this resolution technique is meaningful *only for exponentially distributed service times.*

3.3.5 Results

In this section we validate experimentally the proposed resolution techniques. The accuracy of the client-server analytical models will be compared against the results provided by the queueing network simulator JMT (*Java Modeling Tools [3]*). The simulated network implements the client-server with request-reply behaviour. On the other hand, in the next section we will also show that for "good" interconnection networks the assumption of abstracting away the network from the client-server model is valid.

The test case The modeled scenario has the following features.

- The number of clients is fixed to $p = 16$.

- The average server service time is $T_S = 29\tau$. This value is typical of DRAM2 memories, as we will see in Chapter 5.

- The average process *think* period T_P represents the degree of freedom. The distribution of the *think* periods is exponential. T_P will take its value in the range $[100\tau - 3000\tau]$. Being p fixed, it is necessary to vary T_P in a such a way to emulate all possible load states of the server (unloaded, partially loaded, congested, ...).

- The base latency is $t_{a0} = 72\tau$, which is close to the one of the multi-core architecture that we will study in Chapter 5.

Results will be presented in the following order:

1. exponential distribution of the server service time

2. deterministic distribution of the server service time

For each of them, we will show the progress of R_Q as a function of T_P. In a graph, each curve is identified by a name $CS - X - Y$, with $X \in \{\text{Exp, Det}\}$ and $Y \in \{\text{S, FP, SIM}\}$. X indicates the distribution of the server service time. Y indicates the resolution technique, where S stands for the system of equations 3.1, FP for *Finite Population* (the equation 3.6) and SIM represents the simulation.

Preliminary observations Figure 3.6 shows the JMT simulations of the client-server system (for both the exponential and the deterministic server). It shows the progress of R_Q as function of T_P. Before validating the analytical models against this curve, we should point out the following elements.

- In our test case, where the number of clients is $p = 16$, the effect of T_P is meaningful for fine grain computations ($T_P < 1000\tau$). For larger values of T_P, the under-load memory access latency tends to the base one, since the server is on average unloaded. Obviously, if p grows, even values of $T_P > 1000\tau$ become significant. This behaviour is captured by systems of equations 3.1 and equation 3.6.

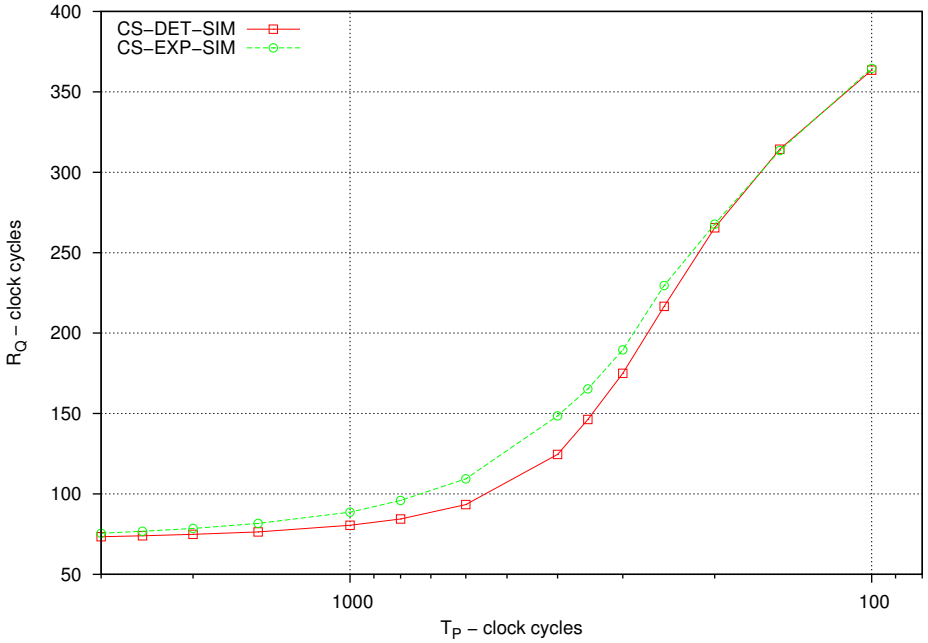


Figure 3.6: Under-load memory access latency.

- Remember that p is the number of processes that are sharing a specific memory macro-module, *not* the number of processors N of the architecture. If p is fixed while N grows, then we would have a change only on t_a0. If the interconnection network is sufficiently good, then there will not be significant effect on R_Q.

- The impact of other architectural parameters "hidden" in the model can be meaningful. For example, T_P depends not only on the sequential algorithm, but even on the cache block size σ. If the memory bandwidth is sufficiently high, large value of σ can be exploited to proportionally reduce the overall number of remote accesses. In this perspective, it is necessary to remark the importance of wormhole flow control that should be provided by the interconnection network.

Comments For an exponential server:

- Figure 3.7 shows the progress of R_Q for both the simulation and the analytical resolution techniques.

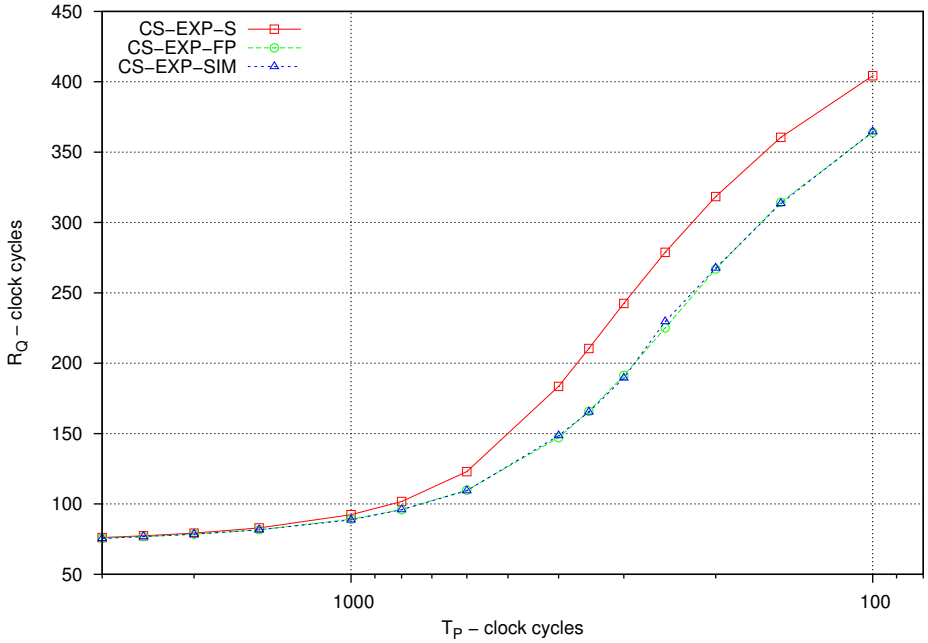
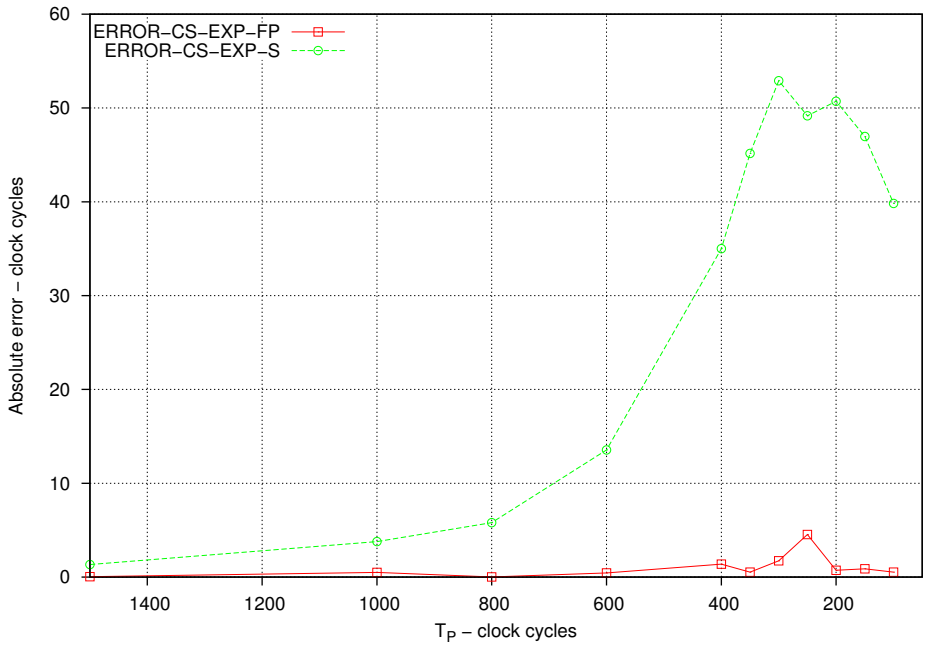


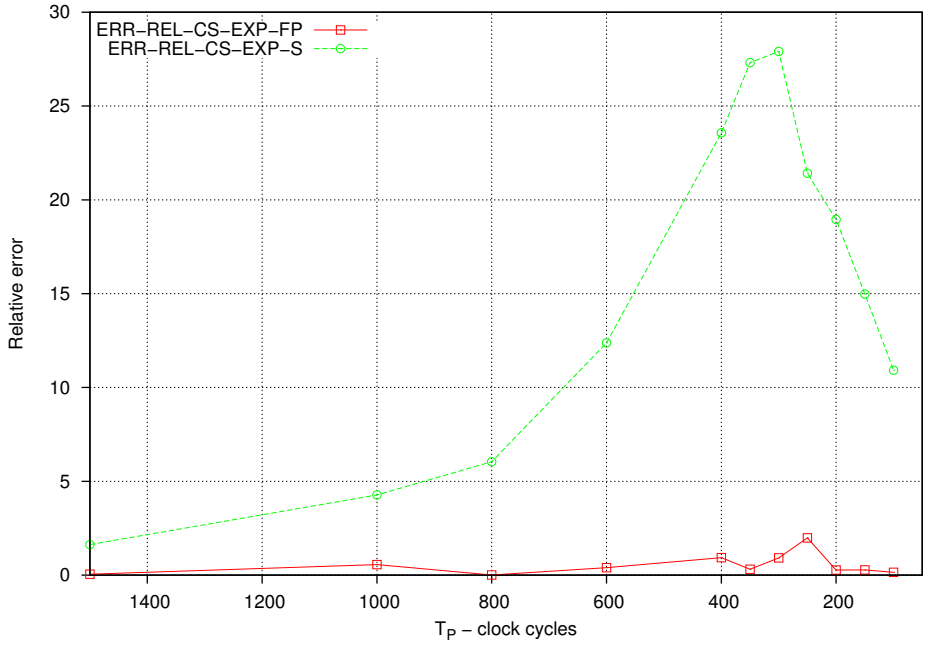
Figure 3.7: R_Q progress obtained by simulation and analytical techniques for an exponential server.

- Figure 3.8 shows the absolute and the relative error (in clock cycles τ) of the analytical resolution techniques with respect to the simulation. Errors become significant as soon as the client requests frequency $\frac{1}{T_P}$ become relevant. Therefore, these graphs are shown in the smaller range $[1500\tau - 100\tau]$.

Analogous graphs are shown for a deterministic server in Figures 3.9 and 3.10.



(a) Absolute error.



(b) Relative error.

Figure 3.8: Errors of resolution techniques for an exponential server.

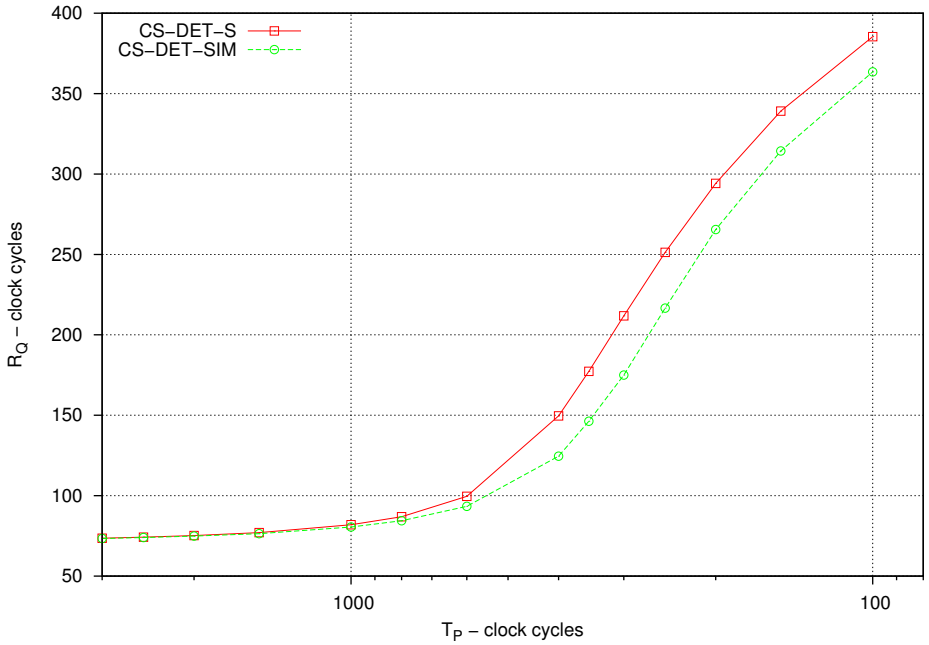
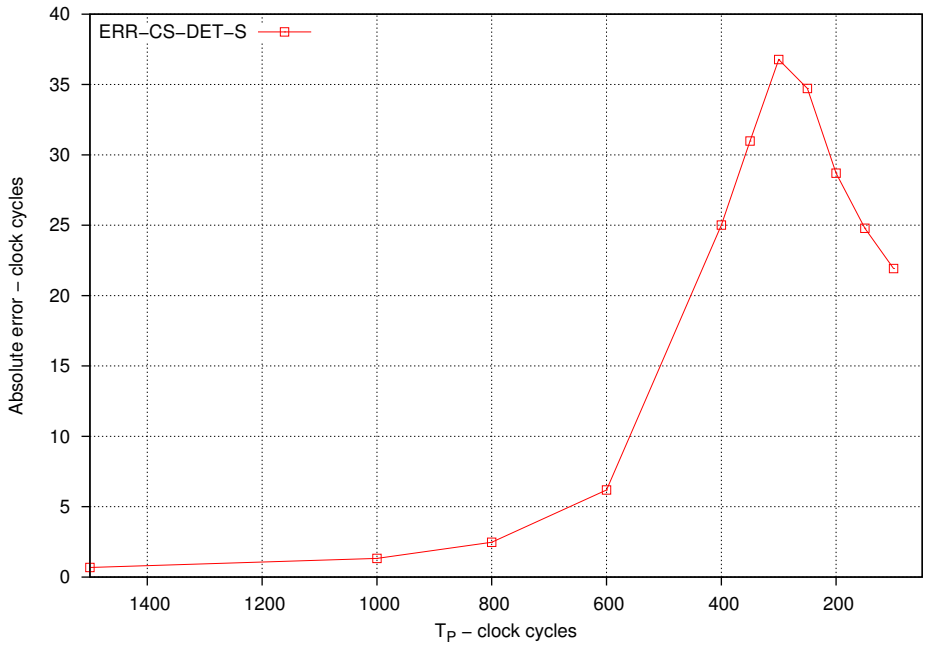
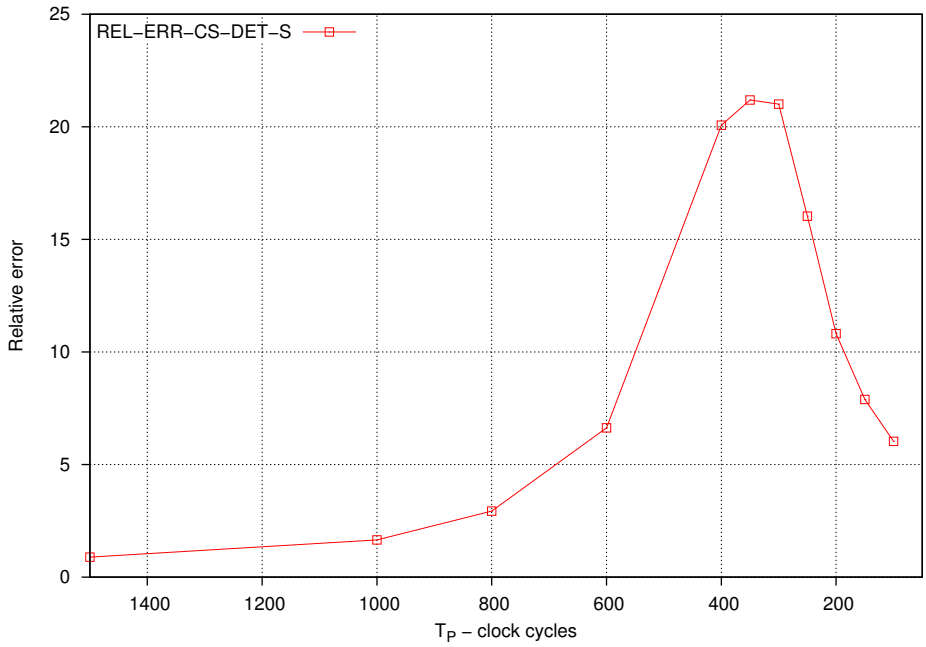


Figure 3.9: R_Q progress obtained by simulation and the analytical technique for a deterministic server.



(a) Absolute error.



(b) Relative error.

Figure 3.10: Errors of the resolution technique for a deterministic server.

In light of these results, we conclude that:

1. $CS - X - S$ ($X \in \{\text{EXP, DET}\}$) techniques provide good approximations of R_Q for high values of T_P, that is when the server is unloaded. $CS - DET - S$ provides a quite good approximation also for low values of T_P (i.e. congested server). However, in both cases the curve progress is not properly modeled whereas the server *begins* congesting (the "critical interval" in which R_Q starts growing significantly). In these cases, the percentage error can exceed the 20%, with peaks approaching the 30% in case of an exponential server.

2. $CS - EXP - FP$ solves this problem for exponentially distributed server service times. In general, $CS - EXP - FP$ almost exactly matches the simulation. Figures 3.7 and 3.8 show that this technique lowers the relative error up to a maximum of 2%.

3.3.6 On the potential impact of the interconnection network

Defining the client-server model we claimed that very often it is reasonable to abstract from the network overhead (due to contention of routers and switches). This is especially true for *fat-tree*-like and on-chip interconnection networks (except buses); see Section 2.2 for more details. In this section we address this topic by referring to a meaningful scenario.

The parameters (service times, number of clients, etc.) of the test case are identical to the ones of the previous tests. We will compare two different simulations:

- one is the classic client-server JMT simulation;

- the other has been obtained using EQNSim [6], a simulator for Extended Queueing Networks (EQN), developed at the Parallel Architecture Lab of the Department of Computer Science, University of Pisa. An EQN enhances the semantics of standard queueing networks by introducing the possibility of modeling aspects proper of elaboration systems. EQNs can be studied only by means of simulation, since both

analytical and numerical resolution techniques cannot be determined, in general, due to the excessive complexity of the model.

For practical reasons (see also Chapter 5) we implemented a simulator of the *Tilera TILE64* multi-core architecture. In this Chip MultiProcessor, 64 cores are interconnected by means of a two-dimensional mesh. Requests toward the memory interfaces may experience the overhead due to network conflicts.

Figures 3.11 and 3.12 show the obtained results. Since the relative error is at most 3%, it is clear that the overhead of the network can be neglected. The reasons are quite intuitive: since the on-chip network is extremely fast with respect to the memory time service (routing of a flit from a switch to another one takes only 1τ), conflicts tend to concentrate only nearby the memory system.

3.3.7 Conclusions

The estimation of the under-load memory access latency is the first step in order to derive the abstract architecture cost model (T_{send} and T_{calc}), by means of which the physical system can be completely abstracted. We will not go into the details of the formal derivation of these functions. The interested reader is invited to consult [20] for more details. However, it is sufficient to know that both T_{send} and T_{calc} are functions of R_Q, therefore a meaningful estimation of its value is crucial.

We have shown that a solution based on pure Queueing Network Theory is unfeasible from the complexity point of view. We prefer the client-server model and its simple analytical resolution techniques based on a minimal set of Queueing Theory concepts. With this approach we enable also qualitative analysis, which is extremely important in our context: for example, we can understand the asymptotic behaviour of a certain metrics (e.g. T_{cl}, R_Q, ...) as a function of other model parameters (e.g. T_P) by taking into account the feedback effect of the system. The importance of qualitative reasoning has been shown in [20].

The original client-server model of [20] has been modified to improve the quantitative analysis. According to the simple intuition of finite population, the assumption of exponential interarrival times has been removed. It has

been shown that the new resolution technique matches the simulation with a maximum relative error of 2%. Unfortunately, this technique holds only for exponential servers.

Although the quality of the results is quite good, in order to apply this model to real architectures we need to take care *at least* of the following aspects:

1. the client-server model cannot be applied to hierarchical systems as it stands;

2. the structure and the specific characteristics of the parallel application are not taken into account yet;

3. modeling queues different than the basic ones ($M/M/1$, $M/D/1$) is quite difficult with this resolution technique. For example, as we will see in Chapter 5, there are some kind of memories that exhibit a non-trivial load-dependent behaviour.

In light of these elements, we advocate that it is necessary to further extend the model and, consequently, the resolution techniques. Unfortunately, the complexity of the problem notably increases. Therefore, in the next chapter we will study the potential advantages coming from the employment of a new modeling technique. It is left as an open problem to find out *approximate* yet *meaningful* analytical resolution techniques modeling the aforementioned topics.

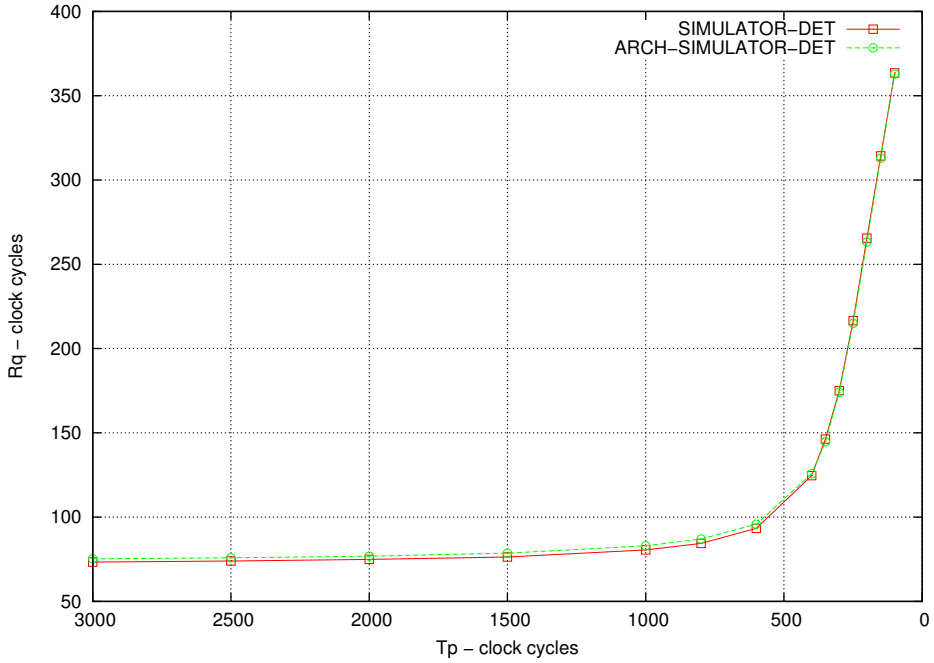


Figure 3.11: The R_Q progress

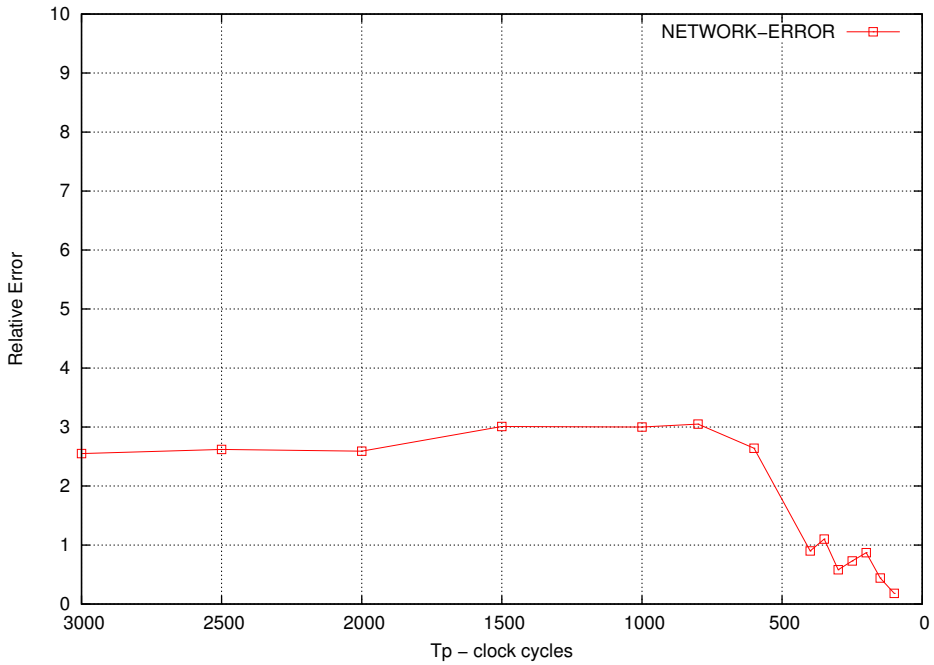


Figure 3.12: Relative error between the two simulations

Figure 3.13: Evaluating the impact of an on-chip interconnection network in a client-server model.

Chapter 4

Stochastic process algebra formalization of client-server models

In the previous chapter we pointed out the necessity of extending the client-server model in the following directions:

- **Probability distributions and queue types.** Interarrival and service times at a queue of a client-server system may not be exponentially distributed. However, it is well-known that providing analytical resolution techniques for non-markovian system is a non-trivial task. We have been able to find a good approximation for deterministic service times in Chapter 3, but what about the modeling, for instance, of deterministic interarrival times? Moreover, the need of modeling different service disciplines at a queue may arise, e.g. queues exhibiting a load-dependent behaviour, as we will see in Chapter 5.

- **Application model.** In the classic model, clients are assumed homogeneous with a fixed ideal interdeparture time T_P. However, clients can behave differently each other. Further, a client itself may exhibit a particularly complex behaviour. For example, a process may alternate different computational *phases*. A phase is characterized by its own specific probability distribution of issuing memory requests. In the simplest scenario a phase of sequential elaboration is followed by a phase of communication (either point-to-point or collective); it is clear

that the load generated on the memory system should be properly modeled according to the phase characterization.

- **Hierarchical systems.** Hierarchical architectures are here to stay. In these systems more than one level of memory hierarchy is shared by the processing nodes (e.g.: shared level-2 caches). Conflicts for accessing shared resources may become significant for what concerns the under-load memory access latency. Somehow, we need to measure these conflicts; perhaps enhancing the client-server structure to model a hierarchy of servers (*hyerarchical client-server systems with request-reply behaviour*).

It is obvious that if we want to take care of these aspects, resolution techniques must be adequately improved. Again, the problem is to determine a trade-off between the complexity of the resolution technique and the quality of the approximation. In light of this, the following methodology is proposed:

- the client-server model with request-reply behaviour remains the reference paradigm (where needed, servers may be structured on a hierarchy),
- but *numerical* resolution techniques will be used to evaluate the under-load memory access latency, in place of the analytical ones.

We advocate that the employment of numerical techniques can overcome the complexity deriving from the formalization of analytical ones. The idea is to describe the client-server model at the level of Markov Chains. There are a lot of resolution methods for moderately sized Continuous Time Markov Chain (CTMC) models, while iterative techniques exist for huge sized models [10]. Since Markov processes can be difficult to construct, we will exploit, as intermediate description language, a *stochastic process algebra* (SPA). An SPA approach is very intriguing because the aforementioned aspects may be addressed with a formal and structured approach.

The modeling of hierarchical systems and an in-depth analysis of the parallel application impact have been addressed, following the new SPA approach, in [5]. On the other hand, in this document we will focus on two other concrete advantages:

1. *from the model point of view*, an extremely simple solution to integrate load-dependent queues;

2. *from the quantitative analysis perspective*, we will see that results obtained with the numerical resolution technique are better than the ones obtained by means of the analytical techniques of Chapter 3.

The structure of the chapter is the following:

1. firstly, we introduce and describe the stochastic process algebra PEPA;

2. secondly, we show how to express a basic client-server model with the new formalism;

3. then the accuracy of the new resolution technique will be compared against experimental results;

4. finally, a very simple solution for modeling load-dependent queues is shown.

4.1 PEPA: a process algebra for quantitative analysis

Performance Evaluation Process Algebra [11] (*PEPA*) is a high-level description language for Markov processes which belongs to the class of *Stochastic Process Algebras* [7] (SPA). Among the wide class of SPAs, we chose PEPA because it is *simple* but at the same time it has sufficient expressiveness for our purposes. The simplicity comes from the structure of the language: PEPA has only a few elements and a formal interpretation of all expressions can be provided by a structured operational semantics. In this section we introduce the minimal set of PEPA features strictly necessary to model client-server with request-reply behaviour systems; for a deeper understanding the reader is invited to consult [11].

The quantitative analysis of PEPA models is based on Markovian Theory. A Markov process relies on the memoryless property of the exponential distribution. The following definition is particularly important.

Definition 4 ***Markov Process.*** *A stochastic process $X(t)$, $t \in [0, \top)$, with discrete state space S is a Markov process if and only if, for $t_0 < t_1 < ... < t_n < t_{n+1}$, the joint distribution of $(X(t_0), X(t_1), ..., X(t_n), X(t_{n+1}))$ is such that*

$$Pr(X(t_{n+1} = s_{i_{n+1}} | X(t_n) = s_{i_n}, ..., X(t_0) = s_{i_0}) =$$

$$Pr(X(t_{n+1} = s_{i_{n+1}} | X(t_n) = s_{i_n})$$

Intuitively, this means that the probability of X to go into the state $s_{i_{n+1}}$ at time t_{n+1} is *independent* of the behaviour of X *prior* to the instant t_n. It is important to keep in mind the memoryless property when working with PEPA.

4.1.1 The PEPA language

A PEPA system is described as the composition of *components* that undertake *actions*. Components correspond to identifiable parts in the system. For instance, in our context, clients and servers will be components of the systems. A component may be atomic or may itself be composed by components. The language is indeed *compositional* in sense that new components may be formed through the cooperation of other ones. Each component can perform a finite set of actions. An action has a duration (or delay) which is a random variable with an *exponential distribution*. Consequently, the *rate* of the action is given by the parameter of the exponential distribution. For example, the expression

$$P \stackrel{def}{=} (\alpha, r).Q$$

represents the definition of a new component P which can undertake an α-action at rate r to evolve into another component Q (defined somewhere else). Since the duration of all actions of the system are exponentially distributed, it is intuitive to say that the stochastic behaviour of the model is governed by an underlying CTMC.

The syntax of the PEPA language is formally defined by the following grammar.

$$\begin{array}{rcl} S & ::= & (\alpha, r).S \mid S + S \mid C_S \\ P & ::= & P \underset{\mathcal{L}}{\bowtie} P \mid P/L \mid C \end{array}$$

S denotes a *sequential component* and P denotes a *model component* which executes in parallel. C and C_S stand for constants to denote either a sequential or a model component. The effect of the syntactic separations is to allow to build only components which are cooperation of only sequential components. This structuring is a necessary condition for building *ergodic* Markov processes, i.e. processes amenable to steady-state analysis [11].

Prefix

$$\frac{}{(\alpha, r).E \xrightarrow{(\alpha,r)} E}$$

Choice

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E + F \xrightarrow{(\alpha,r)} E'} \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E + F \xrightarrow{(\alpha,r)} F'}$$

Cooperation

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E' \underset{L}{\bowtie} F} (\alpha \notin L) \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E \underset{L}{\bowtie} F'} (\alpha \notin L)$$

$$\frac{E \xrightarrow{(\alpha,r_1)} E' \quad F \xrightarrow{(\alpha,r_2)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,R)} E' \underset{L}{\bowtie} F'} (\alpha \in L) \quad \text{where } R = \frac{r_1}{r_\alpha(E)} \frac{r_2}{r_\alpha(F)} \min(r_\alpha(E), r_\alpha(F))$$

Hiding

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\alpha,r)} E'/L} (\alpha \notin L) \qquad \frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\tau,r)} E'/L} (\alpha \in L)$$

Constant

$$\frac{E \xrightarrow{(\alpha,r)} E'}{A \xrightarrow{(\alpha,r)} E'} (A \stackrel{def}{=} E)$$

Figure 4.1: Structured Operational Semantic of PEPA.

The structured operational semantic is shown in Figure 4.1. Below an intuitive description of most used PEPA operator is provided. For a complete treatment the reader is invited to consult [11].

- **Prefix ($(\alpha, r).P$)** This is the basic mechanism to express a sequential behaviour in PEPA. As already said, a component performs an α-action with rate r and it subsequently behaves as P.

- **Choice ($P + Q$)** It represents a component that may behave either as P or as Q. Assume that α and β are the actions that enable respectively P and Q, characterized by their own rate. The idea behind the Choice operator is that once an action has been completed, the other is discarded. For instance, if the first action to be completed is β then the component moves to Q, "forgetting" the other branch.

- **Cooperation ($P \underset{\mathcal{L}}{\bowtie} Q$)** It denotes the cooperation between P and Q over L. L is the cooperation set that contains those activities on which the components are *forced* to synchronize. The rate of this shared activity has to be altered to reflect the slower component in the cooperation (see how in Figure 4.1). It is important to notice that, for actions *not* in L, components proceed *independently* and *concurrently* with their enabled activities. Actually, cooperation is a *multi-way synchronization* since more than two components are allowed to jointly perform actions of the same type.

  When concurrent components do not have to synchronize the cooperation set L is empty; in these cases we will use the abbreviation $P||Q$. We will use also a simple syntactic shorthand to denote an expression like $(P||P||...||P)$ as $P[N]$, with N the number of times that P is replicated.

  Finally, we point out that there can be situations in which two components do synchronize, but the rate of the shared activity is determined by only one of the component in the cooperation. In this case the other component is defined as *passive*. The rate of the activity for the passive component will be denoted with the symbol $\top$.

4.1.2 On the resolution of PEPA models

Solving a PEPA model means solving the underlying ergodic CTMC, i.e. computing the steady-state. We wrote and solved PEPA models using a classic tool like PEPA Workbench [19]. This tool provides a lot of different numerical resolution techniques to solve the model. Different techniques can be employed depending on the size of the resulting CTMC: if the number of states is huge (hundreads of thousands) iterative yet approximate techniques are preferred. Anyway, the models that we treat are extremely small (they never exceed a hundred of states) thus the steady-state has been directly computed employing a very standard algorithm. In all other cases, e.g. when the number of clients significantly grow, a phenomenon known as *state space explosion* may arise. However, thanks to the natural structure of our models, we may take fully advantage from both *state-reduction* and *fluid-approximation* techniques [9]. Briefly, these techniques aim at solving the state space explosion by exploiting potential symmetries in the CTMC. The presence of symmetries can be informally deduced by looking at the PEPA expressions: for instance, in our models a set of homogeneous clients ("$Client[p]$") will induce replicated sub-Markov chains in the underlying CTMC. These replicated subsystems are exploited to restructure the CTMC itself and lowering the state space size. Finally, notice that reduction techniques are automatized in tools like PEPA Workbench.

4.2 Fitting general distributions in PEPA terms

The exponential distribution is not always the most realistic event duration distribution when modeling a system architecture. One critical example involves a memory system that is able to retrieve a requested block in a fixed (constant) amount of time.

Therefore we need a way to fit general (at least deterministic) distributions in a PEPA model, which unfortunately is completely based on CTMC (i.e. on exponential timings). A possible strategy is to make use of *phase type distributions* to approximate general distributions in a standard PEPA model [13]. Phase type distributions are particular distributions constructed by combining multiple exponential random variables. These can be used to

approximate most general distributions. In the following we will concentrate on approximating a deterministic distribution.

The *Erlang* distribution is a commonly used example of a phase type distribution which consists of an exponential distribution repeated k times [16]. The probability density function for the Erlang distribution is as follows:

$$f(x) = \frac{\lambda^k \; x^{k-1} \; e^{-\lambda x}}{(k-1)!}$$

Notice that for $k = 1$ it reduces to an exponential distribution. In PEPA this is generally modelled as a ticking clock [13]:

$$\begin{array}{rcll} Clock_i & \stackrel{def}{=} & (tick, t).Clock_{i-1} & : 1 < i \leq k \\ Clock_1 & \stackrel{def}{=} & (event, t).Clock_k & \end{array}$$

where $t = k\lambda$. *We will use the Erlang distribution to approximate deterministic events*; the greater the value of k (i.e. the more ticks), the closer is the approximation of the Erlang distribution to a deterministic delay. Clearly the value of k cannot be chosen at will, otherwise the number of states in the underlying CTMC could grow significantly. Fortunately, we will see that a value of k is in the range $[2, 4]$ is sufficient to model a deterministic server.

4.3 A PEPA model of client-server systems with request-reply behaviour

4.3.1 The general model

A PEPA model for a client-server system with request-reply behaviour is shown below.

$$\begin{array}{rcl} Client_{think} & \stackrel{def}{=} & (request, r_{request}).Client_{wait} \\ Client_{wait} & \stackrel{def}{=} & (reply, \top).Client_{think} \\ \\ Server & \stackrel{def}{=} & (request, \top).Server + (reply, r_{reply}).Server \\ \\ Client_{think}[p] & \underset{request,reply}{\bowtie} & Server \end{array}$$

Each client (process) operates forever in a simple loop, completing in sequence the two phases *think* and *wait*. The *request* action (phase *think*)

is a shared action between clients and server. It models the situation in which a client sends a request and the server receives it. On the other hand, the time needed to complete the *reply* action (phase *wait*) is *unspecified*, i.e. it will be imposed in another PEPA expression through the cooperation with another component. Therefore, *Client* components see *reply* as a pure synchronization operation.

The server (memory macro-module) can either accept a request from one of the p clients (action *request*) or send them a *reply*. The time to complete a *request* action is obviously unspecified, because it depends only on the clients. The action *reply* is shared to model the fact that a client can go back to the *think* phase as soon as the server has handled its request.

Finally, the last expression instantiate a client-server model with p clients. Notice that the cooperation set contain both the two shared actions *request* and *reply*.

It is useful to highlight that even simpler solutions could be formalized: for instance, the synchronization on the action *request* is not strictly necessary. However we decided to keep it for two reasons. First, it helps at understanding the semantics of the whole system (the "request-reply behaviour"). Second, it will be necessary anyway in further extensions of this basic model.

4.3.2 Applying the model to the processors-memory system

To instantiate the PEPA client-server model on a processors-memory system we need to know the usual parameters.

- T_P is the mean time between two consecutive accesses of a processing node to the same memory macro-module;
- T_S is the average service time of the memory macro-module;
- p is the average number of processing nodes accessing the same memory macro-module.

It is also useful to write the base latency t_{a0} as follows:

$$t_{a0} = T_{req} + L_S + T_{resp}$$

That is, we identify the *request phase req*, the latency of the memory service L_S and the *response phase resp*. Both T_{req} and T_{resp} can be easily determined following the approach of Section 2.2.

With these parameters, the PEPA model can be instantiated as follows:

$$r_{request} = \frac{1}{T_P + T_{req} + T_{resp}}$$

$$r_{reply} = \frac{1}{T_S}$$

4.3.3 Model resolution

The steady-state analysis of a PEPA model gives access to:

- the average population in each state of the underlying CTMC;
- the throughput of the actions.

To determine the average response time of the memory macro-module $R_{Q_{server}}$ it is sufficient to know:

- the average number of clients p_{wait} (out of p) that reside in the $Client_{wait}$ state;
- the throughput λ_{reply} of the action *reply*.

Applying the Little's law (2) we can extract the average time that a client spends in the $Client_{wait}$ state, *that actually corresponds to* $R_{Q_{server}}$:

$$R_{Q_{server}} = \frac{p_{wait}}{\lambda_{reply}}$$

It is extremely important to notice that $R_{Q_{server}}$ *is not the under-load memory access latency* R_Q*, rather it is the average time spent by a request at the memory macro-module.* However, to find out R_Q it is enough to take into account the latency of the request phase T_{req} and the latency of the reply phase T_{resp}. Finally, we end up with

$$R_Q = T_{req} + R_{Q_{server}} + T_{resp} \tag{4.1}$$

4.4 Quantitative comparison with respect to other resolution techniques

Results To evaluate the accuracy of the PEPA client-server model we repeat the same kind of test we did for analytical resolution techniques (Section 3.3.5). Therefore we will compare:

- results of simulations performed with the JMT queueing networks simulator [3], by means of which it has been emulated a client-server system;
- results of the PEPA model;
- results obtained with the classical client-server analytical technique. In particular, we will show again the results obtained by means of the system of equations 3.1 of Section 3.3.3 (In the graphs, the curve name is $CS - X - S$, where X denotes the distribution of the server service time, that will be either exponential or deterministic).

The parametrization of the client-server system is identical to the one adopted in Section 3.3.5:

- The number of clients is fixed to $p = 16$.
- The average server service time is $T_S = 29\tau$. This value is typical of DRAM2 memories, as we will see in Chapter 5.
- The average process *think* period T_P represents the degree of freedom. The distribution of the *think* periods is exponential. T_P will take its value in the range $[100\tau - 3000\tau]$. Being p fixed, it is necessary to vary T_P in a such a way to emulate all possible load states of the server (unloaded, partially loaded, congested, ...).
- The request phase is $T_{req} = 6\tau$ while the response phase is $T_{resp} = 36\tau$, which are close to the ones of multi-core architectures, as we will see in Chapter 5.

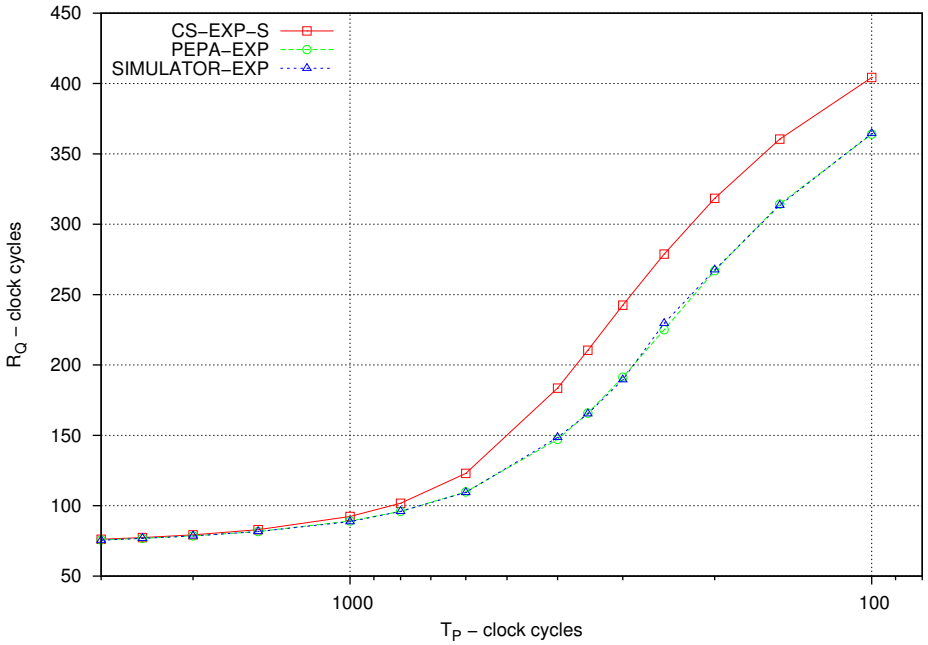
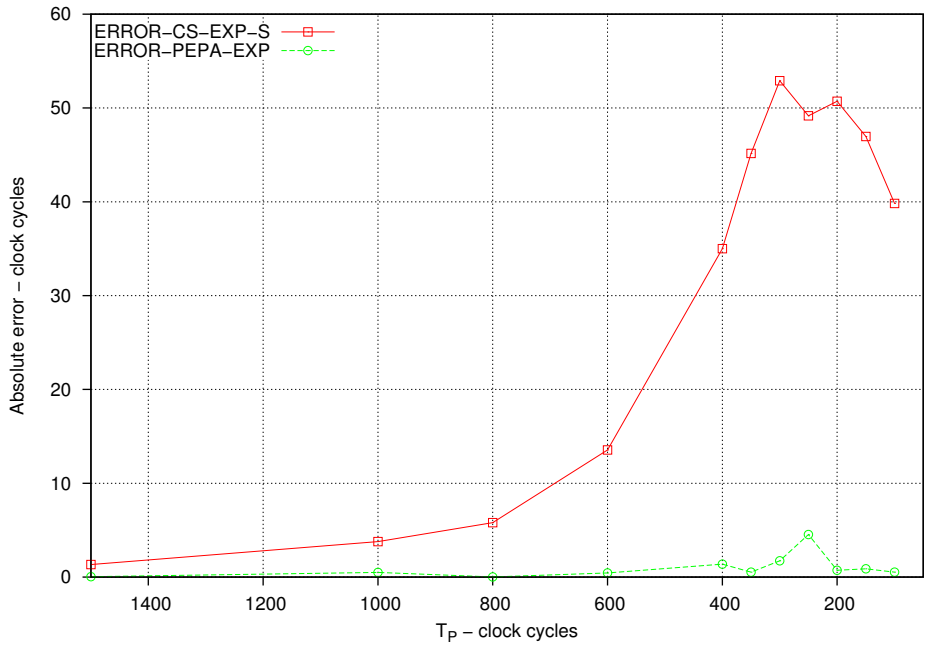
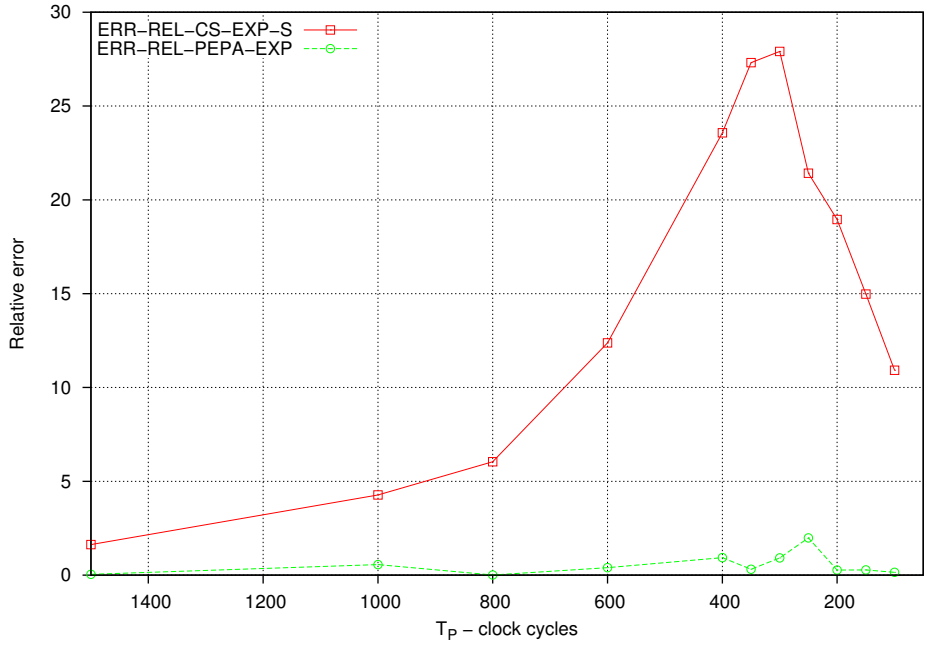


Figure 4.2: R_Q progress obtained by simulation and by analytical and numerical techniques for an exponential server.



(a) Absolute error.



(b) Relative error.

Figure 4.3: Errors of resolution techniques for an exponential server.

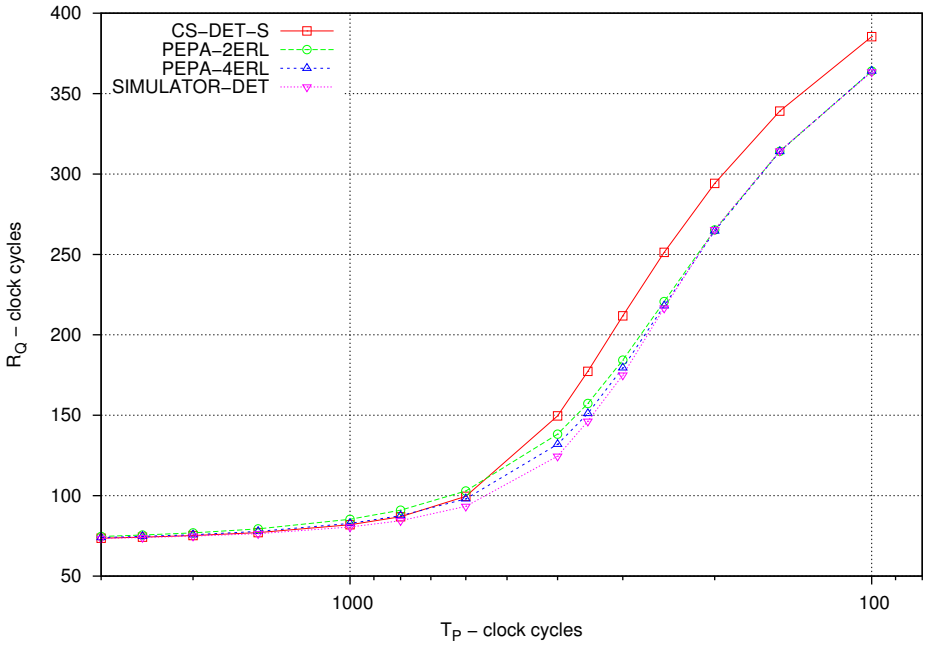
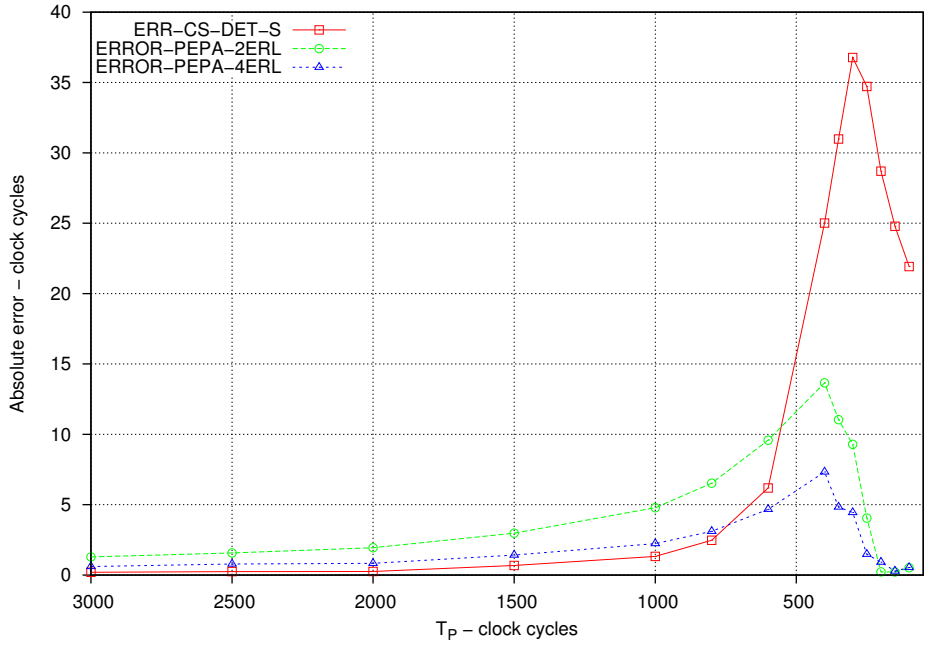
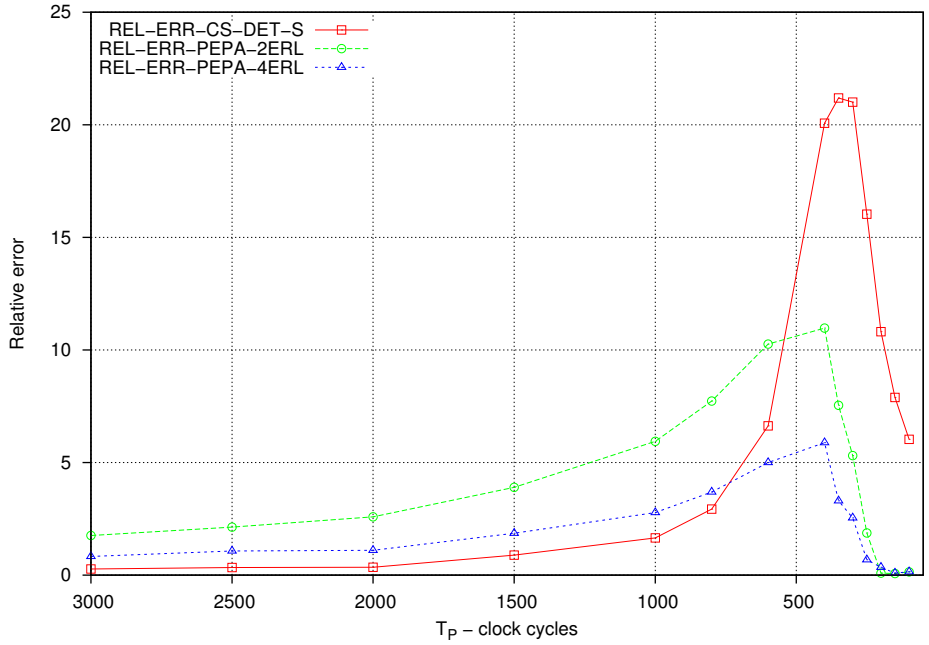


Figure 4.4: R_Q progress obtained by simulation and by analytical and numerical techniques for a deterministic server.



(a) Absolute error.



(b) Relative error.

Figure 4.5: Errors of resolution techniques for a deterministic server.

For an exponential server, Figure 4.2 shows the progress of R_Q in the range of T_P $[3000\tau - 100\tau]$. Figure 4.3 shows the estimation error of each resolution technique with respect to the simulation. Same curves are shown for a deterministic server in Figure 4.4 and 4.5. The deterministic behaviour is implemented in the PEPA model by means of a $k - Erlang$ distribution, $k \in \{2, 4\}$.

Comments The results of the quantitative analysis are in general fairly good. Figure 4.3 states that, for an exponential server, the PEPA approximation matches the simulation with a maximum relative error of 2%. Figure 4.5 shows that approximating the deterministic server service time with a 4-Erlang distribution leads to a maximum percentage error of roughly 6%. Hence, the overestimation experienced with $CS - X - S$ for loaded servers can be definitely overcome with PEPA models.

4.5 An example: load-dependent service times in a client-server model

In this section we propose a PEPA model of a *load-dependent server*. This solution will be exploited in Chapter 5 where a cost model for a real architecture will be derived according to the methodology of client-server systems.

Load-dependence implies that the offered service time varies basing upon the number of requests in the queue. In our context, the larger the number of requests in the queue, the *lower* the experienced service time. This behaviour could be dictated by a lot of factors. For instance, before serving the first of the X requests in the queue, the server could sort them according to some particular criterion. This is what happens in some memory systems [21, 2]: since consecutive accesses for "adjacent" pages (the ones that reside on the same memory row) can be served faster, a group of requests can be formerly reordered.

An example of a PEPA model for a load-dependent server is shown below. Notice two key elements: from one hand the *simplicity* of the model in spite of a meaningful change in the semantics of the server; from the other the possibility of inferring easily the semantics of the server directly from the

PEPA expression.

$$
\begin{aligned}
Client_{think} &\stackrel{def}{=} (request, r_{request}).Client_{wait} \\
Client_{wait} &\stackrel{def}{=} (reply, \top).Client_{think}
\end{aligned}
$$

$$
\begin{aligned}
Server_0 &\stackrel{def}{=} (request, \top).Server_1 \\
Server_1 &\stackrel{def}{=} (request, \top).Server_2 + (reply, \mu).Server_0 \\
Server_2 &\stackrel{def}{=} (request, \top).Server_3 + (reply, 2\mu).Server_1 \\
Server_3 &\stackrel{def}{=} (request, \top).Server_4 + (reply, 3\mu).Server_2 \\
Server_4 &\stackrel{def}{=} (request, \top).Server_5 + (reply, 4\mu).Server_3 \\
Server_i &\stackrel{def}{=} (request, \top).Server_{i+1} + (reply, 5\mu).Server_{i-1} \quad : 5 \leq i \leq N
\end{aligned}
$$

$$
Client_{think}[N] \underset{request,reply}{\bowtie} Server_0
$$

N clients can synchronize with a server. Initially the server is empty ($Server_0$). Once a client generates a request, the server moves to $Server_1$ to indicate that one element is being served. If in the meanwhile another request arrives, then the server moves to $Server_2$ (one element is being served, the other is logically queued). The server can move up to $Server_N$, since N is the maximum number of requests in the system. Depending on the number of requests i in the server, the exhibited service time $T_S(i)$ varies. When a request is being served and the queue is empty, the service time is maximum, that is $T_S(1) = \frac{1}{\mu}$. As soon as new requests arrive, $T_S(i)$ decreases (in this *specific* example the decrement is inversely proportional with i) up to $T_S(i) = \frac{1}{5\mu},\ i \geq 5$. The service time stabilizes at this value, even if new requests arrive.

Chapter 5

Applying the cost model to a concrete architecture

We apply the fundamental results of the previous chapters to a concrete architecture. The structure of the chapter is shown below.

1. The *Tilera TILEPro64* multi-core architecture (from now on Tile64) is introduced. The main features of the architecture are described and commented according to the terminology of Chapter 2.

2. The modeling of the memory system is of capital importance. A problem is that technical information available in literature are not sufficient to understand the behaviour of the memory and predict its service time. It will be shown how to derive it following a reverse engineering approach, i.e. analysing the memory traces retrieved by means of the Tile64 profiler.

3. The whole performance evaluation methodology is revisited with respect to a parallel program executed on the Tile64 architecture. We show how to instantiate the client-server model to predict the performance of the application. The predicted under-load memory access latency will be validated against experimental results.

5.1 Tilera TILEPro64 architecture overview

The Tile64 architecture is a NUMA Chip MultiProcessor where multiple two-dimensional mesh networks interconnect 64 processing nodes (also known as

tiles). It is a notable example of *Network Processor*, i.e. a general purpose architecture mainly oriented to network packet processing. Figure 5.1 illustrates the firmware organization of the architecture with emphasis on an individual processing node's structure.

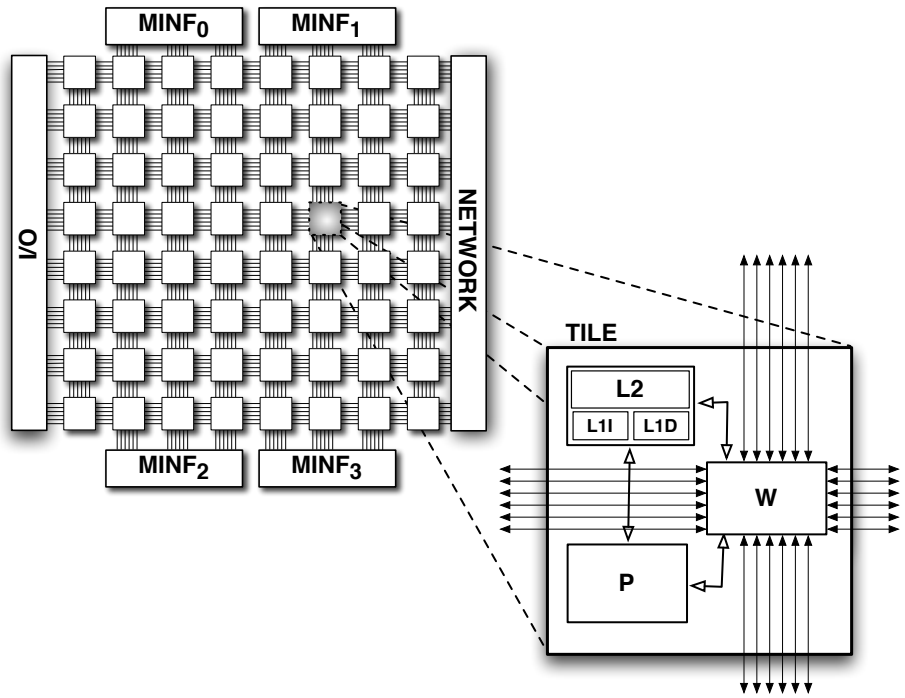


Figure 5.1: The Tile64 firmware architecture

Processing node Each processing node contains a RISC, out-of-order, pipeline CPU P, a private cache hierarchy C and an interface unit W, which is basically a switch implementing deterministic network routing. Notably important is the structure of C. It is a private two-level cache hierarchy: a 16KB $L1$ cache (divided in 8KB *Data* and 8KB *Instructions*) and a 64KB $L2$. This leads to a total of 5MB of on-chip cache. $L1$-*Data* works in write-through mode, while (for logical reasons) $L2$ works in write-back mode. The processing node architecture is therefore extremely simple: in this perspective, notice the lack of both hardware multithreading and a floating point unit.

Interconnection network The five mesh networks are implemented through the W units in each processing node. The flow control is *wormhole*, while the communication protocol between firmware units is based on *timeslot*. Therefore, the latency that a stream of m flits experiences for travelling d units on the network is equal to (see equation 2.2 of Section 2.2):

$$L = (m + d - 2)\ \tau \tag{5.1}$$

Particularly important is the fact that each one of the five mesh addresses a specific functionality. For instance, a mesh MDN is dedicated to the flow of memory access requests/replies between processing nodes and memory interfaces, another one is reserved to cache coherence messages, and so on. One of the important consequence of this structuring is that conflicts on MDN are limited to memory requests and replies. This aspect, together with the fact that MDN is extremely fast if compared to the external memory (see also Section 2.2.2), will allow us to simplify the evaluation of the underload memory access latency by abstracting from the potential overhead of network conflicts.

Cache coherence The Tile64 implements a directory-based automatic cache coherence technique. As we said, bandwidth for cache coherence messages is reserved on an dedicated mesh. However, users are allowed to disable the automatic cache coherence in place of their own algorithm-dependent strategies. This is particularly interesting since provides that flexibility which lacks in many of the state-of-the-art CMPs. In the following, we will not go into the details of cache coherence mechanisms and their potential overhead on the architectural cost model, which is left as an open topic.

Memory system Four memory controllers $MINF_i$ are displaced at two opposite edges of the chip. The role of $MINF$ is to interface the chip with the external memory M, which is a 64-bit DDR2 DRAM. It is worth noticing that the service time of M is *not deterministic*. For instance, *consecutive* requests for adjacent pages, i.e. the ones that reside on the same memory row, are served faster with respect to random ones, since no overhead is paid for the so called "row activation". Locality properties, together with many

others [2], are useful to optimize the global performance of the memory system, from both the bandwidth and the latency point of view. By exploiting these properties, $MINF$ implements a smart scheduling algorithm to reorder outstanding memory requests, instead of forwarding them to M in a classical FIFO manner. The ordered stream of requests should help M in offering a better average service time. Intuitively, the larger the number of outstanding memory requests, the better will be the result of the scheduling algorithm and consequently lower the exhibited service time. As we will see, this behaviour will be modeled representing the memory system as a load-dependent server.

Summary A summary of the Tile64 architectural details is shown in Table 5.1.

CPU clock	800Mhz
L1 size	16KB (8KB D, 8KB I)
L1-Data block size (σ_{L1})	16B
L2 size	64KB
L2 block size (σ_{L2})	64B
W routing latency	1τ

Table 5.1: Architectural characteristics of a Tile64 processing node

5.2 Memory access latency in Tilera architectures

5.2.1 Methodology

The evaluation of the under-load memory access latency R_Q is done according to the methodology of Chapter 4. Summarizing, it consists of three steps.

1. Firstly, *architecture-dependent* parameters are determined. Basically, we are interested in three parameters: the average service time of the memory system T_S, the base latency of the request phase T_{req} and the base latency of the reply phase T_{resp}.

2. Then, *application-dependent* parameters can be extracted from the specific parallel application. In particular, we understood the importance of knowing T_P and p.

3. Finally, the above parameters can be used to instantiate a *PEPA client-server model*, by means of which R_Q will be derived.

In this section we focus on points 1) and 3), while the analysis of application-dependent parameters will be addressed in the next section.

5.2.2 Base latency

Memory controller delay The internal firmware structure of $MINF$ is particularly complex [2], mainly because of the requests scheduler. However, we have experimentally verified that $MINF$ introduces almost always a *constant* delay to forward either memory requests toward M or memory replies toward the chip (equivalently: queueing overhead at $MINF$ is negligible). In the case of a request, the experienced latency is

$$T_{MINF-REQ} = 2\tau$$

while in the case of a reply we have

$$T_{MINF-RESP} = 41\tau$$

The unbalance in these two latencies could be due to many different elements: for instance, the scheduler overhead is included in the reply phase. Yet another simpler possibility is that it is due to a profiler approximation.

Base latency of request and reply phases T_{req} and T_{reply} are functions of the processing node position (x, y) inside the mesh ($x, y \in \{0...7\}$) with respect to a specific memory interface controller $MINF_z$ ($z \in \{0..3\}$). In the following, we fix x, y and z to exemplify the evaluation approach. We will assume $(x, y, z) = (3, 3, 0)$.

The units traversed during $T_{req}(3, 3, 0)$ and $T_{resp}(3, 3, 0)$ are highlighted in Figure 5.2.

- T_{req} begins in the instant in which $L2$ puts a memory request into the network. The size of the request is

$$m_1 = 4$$

  flits [2]. The phase ends as soon as the request gets to M.

- T_{resp} begins in the instant in which M sends the first word of the reply r. The phase ends as soon as $L2$ receives *the first m_2 flits of r*. Being $H = 3$ the size of the message's header, we have

$$m_2 = \sigma_{L1} + H = 7$$

  This is due to the fact that the stall period of P ends as soon as the word for which a fault has been generated is available in $L1$. Therefore, it is not necessary to wait for the whole $L2$ block carried by a message of size $m = \sigma_{L2} + H$.

From Figure 5.2 we understand that the distance between $L2$ and $MINF_0$, in terms of traversed units, is $d = 6$. Equation 5.1 can be used to determine the pipeline latency for the paths $L2 - MINF_0$ and $MINF_0 - L2$. Adding the latency of $MINF_0$, that is $T_{MINF-REQ}$ for the request phase and $T_{MINF-RESP}$ for the reply phase, we can determine $T_{req}(3,3,0)$ and $T_{resp}(3,3,0)$ as follows.

$$T_{req}(3,3,0) = (m_1 + d - 2) + T_{MINF-REQ} = 10\tau$$

$$T_{resp}(3,3,0) = (m_2 + d - 2) + T_{MINF-RESP} = 52\tau$$

Average values of T_{req} and T_{resp} for a set of tiles S can be also meaningful. They can be derived in a very analogous way, by using the average distance d_{avg}, between tiles of S and a specific $MINF$ unit, in place of d.

Approximation The previous evaluation hides a subtle approximation. Consider the two following transitories.

- At the *beginning* of the *request* phase, *before getting stalled*, P exploits the out-of-ordering technique to execute instructions for a total of X additional cycles.

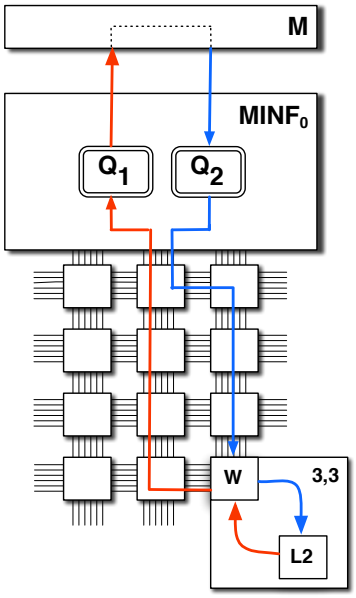


Figure 5.2: Traversed units during the request phase (red line) and reply phase (blue line)

- At the *ending* of the *reply* phase, the first σ_{L1} flits of the block loaded from M have been received by $L2$. Actually, before using the requested word, P remains stalled Y additional cycles waiting for the σ_{L1} words to be transferred from $L2$ to $L1$.

However, we experimentally verified that $X \approx Y$, so the previous evaluation still represents a very good approximation.

5.2.3 Under-load latency

Model of the memory system For the reasons explained in the previous section, the external memory macro-module M *and* its $MINF$ unit can be modeled as a *load-dependent single-server queue*. We will indicate with $T_S(q)$ the average service time when q elements are outstanding in the server. To determine $T_S(q)$, we undertook an accurate experimentation because no technical information is available in literature. We performed a lot of simulations; in each one of them, the memory has been subjected to different workloads. By sampling the queue size during each service, we ended up

with a large set of couples $\langle q, T_S(q)\rangle$. Then, we fixed q and took the average of $T_S(q)$, leading to the values of Table 5.2.

q	$T_S(q)$ (τ)
1	32.41
2	24.49
3	20.61
4	16.88
5	15.43
6	15.15
7	14.26
≥ 8	14

Table 5.2: Service times for the Tile64 memory system

Under-load memory access latency By taking into account the load-dependent nature of the memory system, R_Q can be determined resorting on the PEPA model for load-dependent client-server systems of Section 4.5. We already found the architecture-dependent parameters to instantiate the model, that is T_{req}, T_{resp} and $T_S(q)$. In the next section we will focus on a specific parallel application to exemplify the prediction of R_Q and to validate it against experimental results.

5.3 Performance evaluation of a parallel application

A parallel application will be executed on the Tile64 Architecture Simulator (from now on TAS) [1]. In the following, we will distinguish between the memory latency $R_{Q_{sim}}$ sampled with the TAS profiler and the one predicted by the Tile64 cost model R_Q. The parallel application will be executed for different values of the parallelism degree N. At the end of the test we will compare the progress of $R_{Q_{sim}}$ and R_Q.

Parallel application The application in question implements a farm paradigm. N identical worker processes executes a very classic sequential algo-

rithm, i.e. the *matrix-vector multiplication*. Each worker receives, from an emitter process, a matrix of size $M = 256KB$ bytes (65536 elements of size 4 bytes), then executes the algorithm and finally sends the resulting vector to a collector process.

Sequential performance A parameter we need to derive is T_P, i.e. the average time interval between two consecutive accesses at the same memory macro-module originated by the same processing node. Clearly, T_P is an algorithm-dependent parameter. To determine T_P we use the following formula.

$$T_P = \lceil \frac{Execution\ Time + Stall\ time}{\sharp\ L2\ Faults} \rceil$$

Table 5.3 shows some performance data of the sequential matrix-vector multiplication algorithm, retrieved by means of TAS. Instantiating the previous formula with these data, we obtain

$$T_P = 1054\tau$$

Execution time (not stalled)	3152529τ	L2 Read faults	4149
Pipelining stall	837122τ	L2 Write faults	34
$L1$-Data stall	94767τ	L2 Inst faults	13
$L2$ stall	337210τ		

Table 5.3: Performance data for the sequential Matrix-Vector multiplication executed on the Tile64 architecture.

On the value of p The other important point to discuss is the number p of processor (out of N) that share a memory macro-module. We should find a $low - p$ mapping for *this specific application*, that is a smart strategy of allocating data structures (either shared or private) to memory macro-modules, in such a way to minimize p [20]. Indeed, we know that performance problems, though related to network distance effects, are mainly influenced by the *contention effects*. A possibility is to concentrate the data structures of a worker, both the ones of the program and the ones of the runtime support,

on a single memory macro-module. For instance, if the parallelism degree of the application were maximum, i.e. $N = 64$, we would have $p = 18$ since each of the four memory macro-modules would be shared by 16 workers plus emitter and collector. However, since we are interested in *predicting* performance and not in the performance itself, we are going to assume $p = N$, that is all the processes will share the same macro-module. This is probably the worst strategy from the *performance* viewpoint, but it allow us to analyse $R_{Q_{sim}}$ for more intensive workloads, since p may be chosen at will in the range $[1 - 64]$. Without loss of generality, the data structures of the p workers will be allocated on the macro-module interfaced by $MINF_0$.

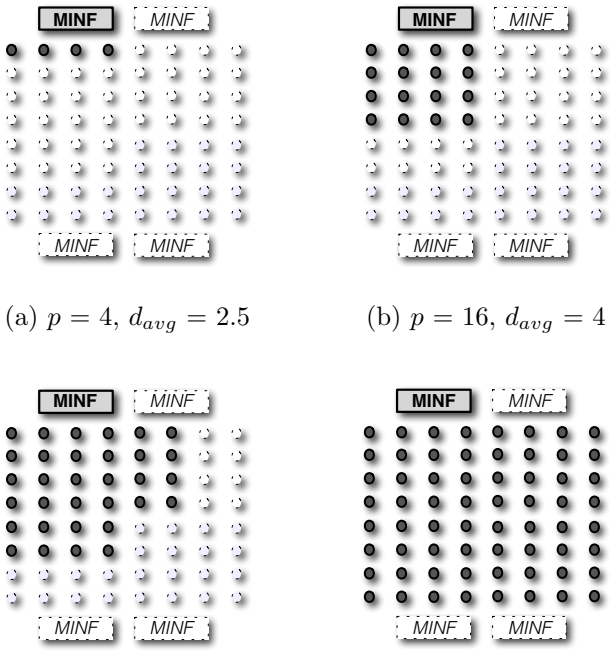
Assumption The parallel application is composed by three kind of processes: emitter, collector and workers. This would imply to instantiate a client-server model with *heterogeneous* clients, that is not the scope of our analysis. This kind of systems has been treated in [5]. To simplify the study, we decided to focus only on worker processes. Hence:

- in the TAS experiments it is assumed the existence of an infinite input stream of matrices with interarrival time such that workers never get stalled waiting for data to elaborate. The collector simply discards the received elements. Therefore, the workload on the memory system is generated by the worker processes.
- in the cost model we will have homogeneous clients (worker processes).

Actually, this is not a restrictive approximation. Especially for high values of N, the impact of both the emitter and the collector become negligible with respect to the one of $N - 2$ workers. This consideration has been experimentally verified: we saw that for $N \geq 16$ there is no meaningful change in the workload generated either by N workers or by $N - 2$ workers plus emitter and collector.

Structure of the test The TAS provides users with a lot of primitives to manage aspects like the program's data structures allocation and the process-to-tile (PTT) mapping. This allows us to set important parameters at will, like p and the average distance between processes and a memory macro-module. The application has been executed for $p \in \{4, 8, 16, 24, 32, 48,$

64}. Some PTT mappings and the relative average distance d_{avg} between tiles and $MINF_0$ are illustrated in Figure 5.3.



(a) $p = 4$, $d_{avg} = 2.5$ (b) $p = 16$, $d_{avg} = 4$

(c) $p = 32$, $d_{avg} = 5$ (d) $p = 64$, $d_{avg} = 7$

Figure 5.3: Process-to-tile mappings.

Results and comments Table 5.4 summarizes constants and parameters to instantiate the PEPA model for a load-dependent client-server system,

T_P	1054τ
T_{req}	$(4 + d_{avg})\tau$
T_{resp}	$(46 + d_{avg})\tau$
$T_S(i)$	load-dependent, see Table 5.2
p	{4, 8, 16, 24, 32, 48, 64}

Table 5.4: Cost model constants and parameters.

Figures 5.4 and 5.5 compare the simulated and the predicted under-load memory access latency, respectively indicated with $R_{Q_{sim}}$ and R_Q. Thanks

to both the fast on-chip interconnection network and the efficient memory system, even for high values of p the experienced $R_{Q_{sim}}$ does not show a dramatic increment with respect to the base latency. Clearly, we should interpret this result in light of both the assumptions we made and the application we are modeling. For instance, we notice the absence of meaningful communication patterns between processes, that could lead to a remarkable degradation of $R_{Q_{sim}}$. Another example is that, for certain algorithms, the out-of-order behaviour of the processing nodes may lead to a significantly lower T_P (and consequently to a greater workload on the memory system) by issuing more than one memory access request before getting stalled. In general, we advocate that even with a little change in either the sequential algorithm or in the parallel paradigm, the overall performance may drastically change.

The Tile64 cost model, at least for this specific application, provides a very good approximation to $R_{Q_{sim}}$. Apart from the asymptotic behaviour of the two curves, that is identical, the maximum absolute error of R_Q does not exceed 10τ, for a corresponding percentage error lower than 10%.

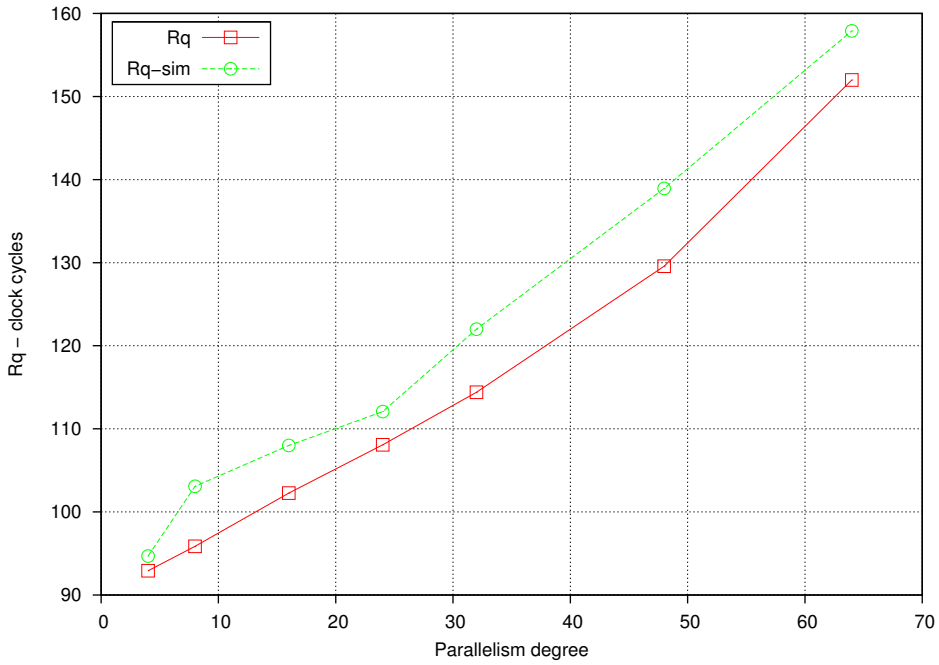
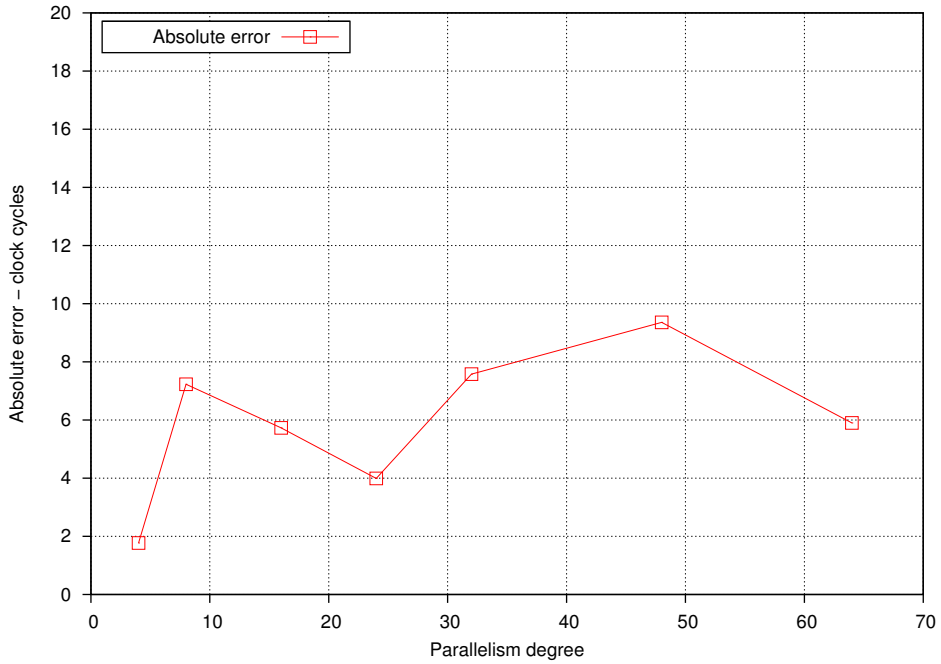
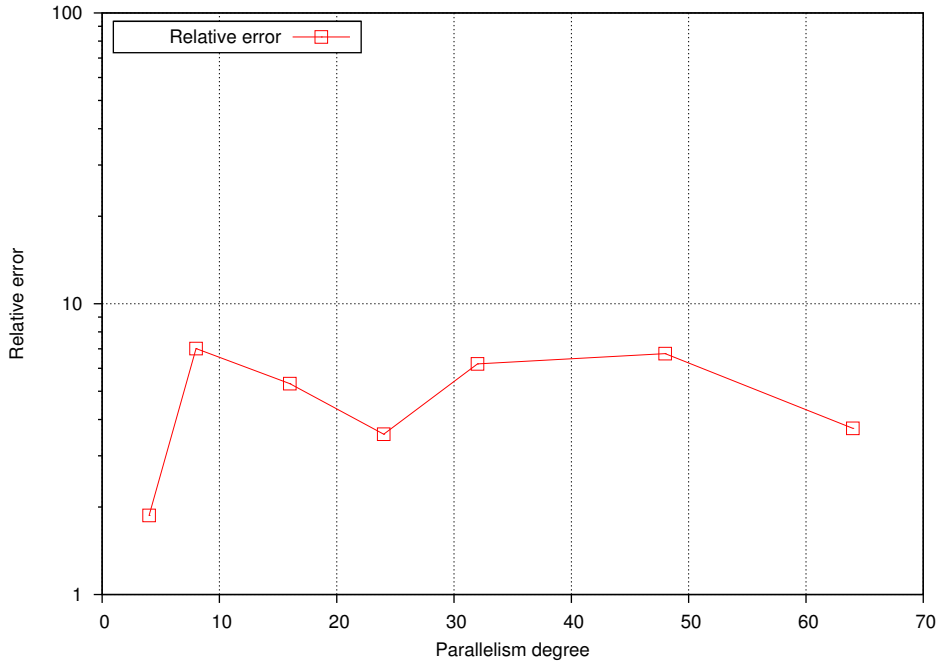


Figure 5.4: Under-load memory access latency experienced with the simulation ($R_{Q_{sim}}$) and predicted with the cost model (R_Q).



(a) Absolute error



(b) Relative error

Figure 5.5: The Tile64 cost model estimation error with respect to the simulation for a matrix-vector multiplication task farm.

Chapter 6

Conclusions

In this chapter the main results of the thesis are summarized and directions for further work are discussed.

6.1 Summary

A lot of architectural choices are possible when building upcoming chip multiprocessors CMP or, more in general, multiprocessors: parallelism degree of the architecture, complexity of the processing nodes itself, interconnection networks, memory hierarchy, cache coherence protocols and many others. In particular, the functional sharing of resources is fundamental to optimize the *global performance of the architecture*; unfortunately, this has also a side effect, that is the complexity of *predicting* the *single flow performance*. In the perspective of structured parallel programming, a *performance cost model* would be of invaluable importance to allow a compiler to evaluate, configure and optimize parallel programs. In this thesis we addressed the problem of formalizing this cost model.

Analytical resolution techniques for client-server models Our starting point was the cost model defined in [20]. As far as we know, it is the only architectural cost model based on the idea of retrieving a precise estimation of the application performance by resorting on a *detailed* and *machine-dependent* prediction. In [20], the shared memory architecture is modeled as a client-server system with request-reply behaviour. The aim of this modeling is to determine the response time of the server, which actually corresponds

to the under-load memory access latency R_Q. By exploiting basic Queueing Theory results, a first analytical resolution technique S is provided. S is an *approximate* resolution technique, i.e. it relies on some assumptions and hypothesis, but it is also extremely *simple* from the mathematical point of view.

The first objective of the thesis was to validate the accuracy of S, with emphasis on CMP architectures. The comparison between simulations and analytical results states that *S overestimates R_Q as soon as the number of conflicts for accessing a memory macro-module becomes significant*. The second goal of the thesis was to overcome this problem. Therefore, we proposed a new resolution technique FP ("Finite Population"). FP exploits the idea that the population in a client-server system is *fixed*, thus the interarrival rate at the server cannot be assumed constant as in S. FP matches the simulation results almost exactly, but it works only for exponential servers, since it is based on pure Markovian Theory.

Numerical resolution techniques for client-server models The modeling of far complex architectures implies changes to the semantics of the client-server system. For instance, we have pointed out the necessity of modeling load-dependent servers and hierarchical architectures; even a less abstract representation of the application workload could provide meaningful improvements, as shown in [5]. In light of these elements, we proposed a new approach to the modeling of such systems, based on a *stochastic process algebra*. We introduced PEPA and its relationship with the Markovian Theory. Therefore, we moved from analytical resolution techniques to numerical ones. A side effect of the PEPA modeling is an improvement to the accuracy of the R_Q prediction. All the work on PEPA has been undertaken according to the classical principles: simplicity of the approach (which explains also the choice of the language) and quality of the approximated results.

Quantitative analysis of client-server systems We validated both analytical and numerical techniques against direct experimentation. Table 6.1 summarizes the results we achieved by showing the maximum percentage estimation error for a not congested memory macro-module.

	Techniques		
	S	FP	$PEPA$
Exponential	23.57%	0.93%	0.93%
Deterministic	20.07%	-	5.88%

Table 6.1: Analytical and numerical resolution techniques for client-server models. In each cell the *maximum relative error* for a not congested server is showed.

The methodology applied to a concrete architecture A cost model for a concrete architecture, the Tile64, has been provided and validated. In this part of the thesis we worked in two directions.

1. The architecture has been analysed to derive constants and parameters of the cost model. An in-depth study of the memory system, as well as network latencies, characterize our work. In this experimentation phase, the Tile64 profiler has been extensively used.

2. A client-server system with load-dependent semantics, expressed in PEPA, has been used to formalize the Tile64 cost model. A simple parallel application has been chosen and algorithm-dependent parameters have been extracted. At this point, we were able to instantiate and validate the Tile64 cost model.

The accuracy of the predicted R_Q turned out fairly good. Clearly, further experiments should be done with different, far complex parallel applications.

6.2 Further work

Guidelines on how to advance the research go in many directions.

1. **Analytical techniques.** The goal is to extend the techniques of Chapter 3 to model those aspects that suggested the employment of PEPA. The problem is always the same: find a *simple* yet *fairly accurate* resolution approach, as the one based on a system of linear equations.

2. **Application's workload.** The cost model we provided for the Tile64 in Chapter 5 should be validated with more complex applications, e.g.

data parallel computations with stencils or even composed skeletons. In this perspective, the cost model may be extended with the techniques formalized in [5], e.g. client-server with heterogeneous clients, computational phases for a more detailed workload model, etc.

3. **Hierarchical systems.** A cost model for hierarchical systems has been already formalized by means of PEPA [5]. The knowledge we matured on the Tilera architecture could be exploited to validate the accuracy of this cost model: upcoming Tilera CMPs are hierarchical systems, though their firmware-assembler architecture resembles the basic one of the Tile64.

4. **Abstract architecture cost model.** Once found, the under-load memory access latency should be used to express the two fundamental functions of our abstract architecture: T_{send} and T_{calc}. Following the methodology of [20] and taking into account the specific features of the run-time support to concurrency mechanisms, the next step is to formalize and validate the accuracy of the aforementioned functions. At a higher abstraction level, the final step would be to instantiate (evaluate, configure, optimize) the paradigms formalized within the context of structured parallel programming on the basis of T_{send} and T_{calc}.

Bibliography

[1] Multicore Development Environment User Guide. Technical report, Tilera Corporation, 2010.

[2] Tile Processor I/O Device Guide. Technical report, Tilera Corporation, 2010.

[3] Java Modelling Tools. Dept. of Electronics and Computer Science, Polytechnic of Milan, 2011.

[4] M. A. Arsan and F. Neri. Elementi di teoria delle code. Lecture notes in Teletraffic Engineering.

[5] A. Bandettini. Cost models for parallel applications on shared memory architectures. Master's thesis, University of Pisa, 2011.

[6] D. Buono. EQNSim. A simulator for Extended Queueing Networks.

[7] A. Clark, S. Gilmore, J. Hillstone, and M. Tribastone. Stochastic process algebras.

[8] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: a Hardware-Software Approach.* Morgan Kaufmann, 1 edition, August 1998.

[9] Richard Hayden and Jeremy T. Bradley. A fluid analysis framework for a Markovian process algebra. *Theoretical Computer Science*, 411(22–24):2260–2297, April 2010. Submitted to TCS, September 2008. Accepted 5 Feb 2010.

[10] J. Hillston. Process algebras for quantitative analysis. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 239–248, Washington, DC, USA, 2005. IEEE Computer Society.

[11] J. Hillstone. *A Compositonal Approach to Performance Modelling*. Cambridge University Press, 1996.

[12] G. Iazeolla. *Impianti, Reti, Sistemi Informatici*. Franco Angeli, April 2004.

[13] A. Argent Katwala, J. T. Bradley, N. Geisweiller, S. T. Gilmore, and N. Thomas. *Modelling Tools and Tecniques for the Performance Analysis of Wireless Protocols*.

[14] L. Kleinrock. *Queueing Systems*, volume 1: Theory. Wiley- Interscience, 1975.

[15] Silvia Lametti. Modello dei costi delle tecniche di cache coherence nelle architetture multiprocessor. Master's thesis, Dept. of Computer Science, University of Pisa, Italy, October 2010.

[16] A. Papoulis and S. U. Pillai. *Probability, Random Variables and Stochastic Processes*. Mc Grow Hill, 4 international edition, 2002.

[17] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: the Hardware-Software Interface*. Morgan Kaufmann, 4 edition, November 2008.

[18] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *Journal of the Association for Computing Machinery*, 27(2):313–322, April 1980.

[19] M. Tribastone, A. Duguid, and S. Gilmore. *The PEPA Eclipse Plug-in. Performance Evaluation Review*, volume 36. 2009.

[20] M. Vanneschi. *Architettura degli Elaboratori, and Course Notes of High Performance Systems and Enabling Platforms, Part 2, Section 3 - Master Program in Computer Science and Networking.* Pisa University Press, 2011.

[21] Wikipedia. en.wikipedia.org/wiki/dynamic_random-access_memory.